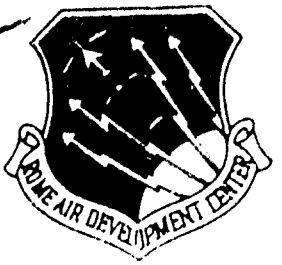
RADC-TR-80-261
Final Technical Report
August 1980

LEVEL II 12



JOVIAL J73 AUTOMATED VERIFICATION SYSTEM - STUDY PHASE

General Research Corporation

Carolyn Gannon

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-80-261 has been reviewed and is approved for publication.

APPROVED: Frank S. LaMonica

FRANK S. LAMONICA
Project Engineer

APPROVED: Wendall C. Bauman

WENDALL C. BAUMAN, Colonel, USAF
Chief, Information Sciences Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed, if you wish to be removed from RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (ISIE) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1 REPORT NUMBER: RADC-TR-80-261	2 GOVT ACCESSION NO: AD-A091190
3 RECIPIENT'S CATALOG NUMBER:	
4. TITLE (and Subtitle): JOVIAL J73 AUTOMATED VERIFICATION SYSTEM - STUDY PHASE	5 TYPE OF REPORT & PERIOD COVERED: Final Technical Report, Sep 79 — Apr 80
	6 PERFORMING ORG REPORT NUMBER: N/A
7 AUTHOR(s): Carolyn Gannon	8 CONTRACT OR GRANT NUMBER(s): F30602-79-C-0265
9 PERFORMING ORGANIZATION NAME AND ADDRESS: General Research Corporation, P O Box 6770, Santa Barbara CA 93111	10 PROGRAM ELEMENT PROJECT TASK AREA & WORK UNIT NUMBERS: 63728F, 25320206
11 CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center, Griffiss AFB NY 13441	12 REPORT DATE: Aug 1980
	13 NUMBER OF PAGES: 93
14 MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Same	15 SECURITY CLASS (of this report): UNCLASSIFIED
	15a DECLASSIFICATION DOWNGRADING SCHEDULE: N/A

16 DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17 DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18 SUPPLEMENTARY NOTES

RADC Project Engineer: Frank S. LaMonica (ISIE)

19 KEY WORDS (Continue on reverse side if necessary and identify by block number)

Computer Software Testing
Computer Software Verification
JOVIAL J73
Software Development Tool
Computer Programs
Software

20 ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report presents the results of a study to specify the required capabilities and high-level design of an automated tool to support the testing and verification of JOVIAL J73 software systems. Included is a state-of-the-art review of software testing and verification with emphasis on techniques applicable to JOVIAL J73 programs.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

402754

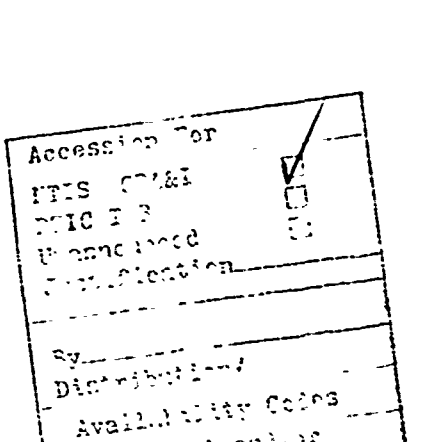
ABSTRACT

This report is primarily a review of the state-of-the-art of software testing and verification with emphasis on techniques applicable to JOVIAL J73 programs. Since the project concerns a JOVIAL J73 Automated Verification System, the need for such a tool, the capabilities for the tool, and the high-level design of the tool are also described. Future capabilities for the tool are identified.

CONTENTS

FIGURES

TABLES



EVALUATION

The purpose of this contractual effort was to determine and specify the required capabilities for an automated testing and verification system for JOVIAL J73 software systems. The effort provided a significant review of the state-of-the-art of software testing and verification, with emphasis placed on techniques applicable to JOVIAL J73 programs. The resulting capabilities were specified in two separate documents - a Functional Description and a System/Subsystem Specification, which will be utilized during the implementation phase of the effort. The availability of an automated testing and verification system for JOVIAL J73 is significant in that it will enhance Air Force software development capability and result in a more cost-effective and reliable product. This effort was responsive to the objective of the RADC Technology Plan, TPO 4G4, "Higher Order Languages."

Frank S. LaMonica

FRANK S. LAMONICA
Project Engineer

1 INTRODUCTION

General Research Corporation is under a two-phase contract with Rome Air Development Center to develop and implement an automated tool to assist in the testing, verification, and maintenance of JOVIAL J73 software. Phase I of this effort is the study of the state-of-the-art of software verification techniques and tools and the development of a functional description and system/subsystem specification for the tool. Phase II of this effort is the implementation, testing, and user training period.

This report describes the need for such an Automated Verification System (AVS), results of the state-of-the-art study, highlights of the functional description and system/subsystem specification, and future capabilities for consideration. Additional reports resulting from this effort are the following:

Phase I

1. Functional Description
2. System/Subsystem Specification
3. Project Resource Document

Phase II

4. User's Manual
5. Maintenance Manual
6. Test Plan
7. Final Report: Implementation Phase
8. Program Specification

The implementation of the AVS, called J73AVS, is expected to commence in May 1980. Figure 1.1 is a schedule of activities for both phases of this effort. Final delivery of J73AVS is scheduled for October 1981 on the Itel AS/5 at Wright-Patterson AFB and the DEC 20 at Rome Air Development Center at Griffiss AFB.

Each three months the developing system will be benchmarked; that is, an execute document (absolute file) will be created containing the current tool. It is expected that the tool will have the following capabilities at each benchmark:

Benchmark 1 - Command and control
Database management
Syntax analysis

Benchmark 2 - Structural analysis
Instrumentation
Static analysis
Reaching set generation

Benchmark 3 - Report generation
Path analysis
Post-execution analysis
Test history processing

The incremental benchmarks are intended for our use of J73AVS to analyze its own code, and for limited use at Wright-Patterson by Government personnel to give the tool early exposure.

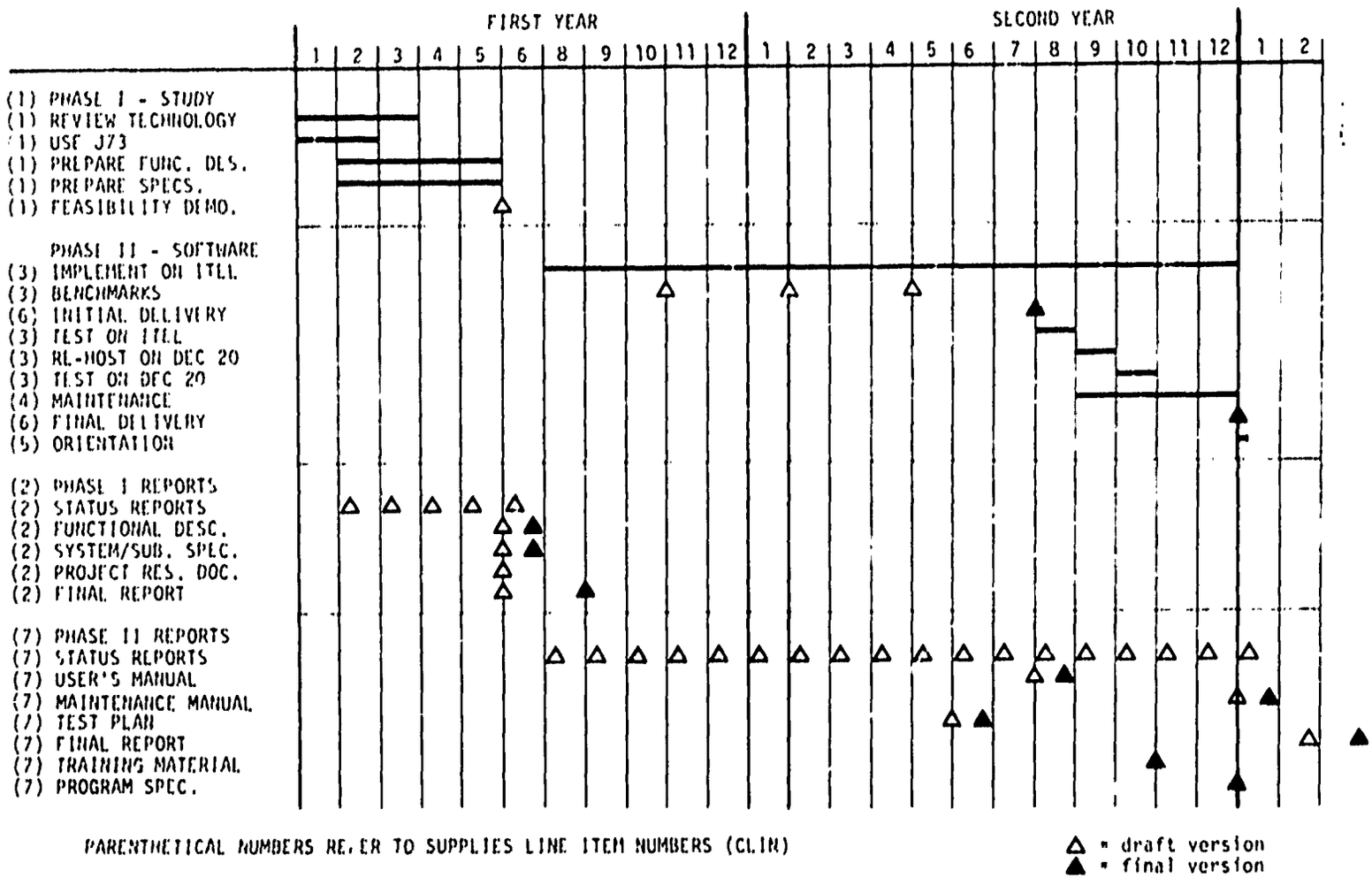


Figure 1.1. Schedule of Deliverables

2 THE NEED FOR J73AVS

The need for this automated verification system is based upon the emergence of a new JOVIAL language which will supersede the previously-approved JOVIAL dialects; the characteristics of the language that make it complex and error-prone; the type of applications expected to be written in the language; and the standardization of certain testing measures.

In an effort to prescribe a standard policy for using computer programming languages and for testing computer programming language compilers, the Air Force issued AF Regulation 300-10 in 1976. Two JOVIAL languages, J3 and J73/1, were specified as Air Force standard high-order programming languages. Both JOVIAL languages are primarily designed for command and control system programming. They are especially well suited to large systems requiring efficient processing of a large volume of data with complex structure.

Another JOVIAL language, J3B, evolved from J3 for the purpose of developing computer programs for the Boeing B-1. Derivatives of J3B have been widely used for avionics computer programming. However, JOVIAL J3B is not a language approved by AF Regulation 300-10. Therefore, a blend of J73/1 and J3B, plus additional features not in either language, has been created to satisfy the programming needs of both the avionics and systems communities. This language, JOVIAL J73, is specified in MIL-STD-1589A and is being refined for a July 1, 1980 release. In the spring of 1980, AF Regulation 300-10 is expected to be revised to cancel both J3 and J73/1 languages, leaving J73 as the only JOVIAL language.

It was the desire to improve software reliability that prompted the Air Force's request for an Automated Verification System (AVS) to be developed and made available as soon as possible following release of validated JOVIAL J73 compilers. Encouragement for an AVS and other

sup[illegible]t tools also came from the JOVIAL Users Group, a body of interested management and technical people from industry, Government, and the Air Force.

2.1 CHARACTERISTICS OF J73 PROGRAMS

As defined in MIL-STD-1589A, JOVIAL J73 permits the independent processing of functional modules which communicate through compools and argument transmission. J73 permits both recursive and reentrant procedures for effective multi-processing. The language provides a rich variety of data types and supporting data manipulation functions, making assembly code programming unnecessary for most applications. However, except for a trace directive which supplies limited output facility, there is no input/output capability in the language. Linkages to assembly or alternate-language routines are required for input and output.

Storage allocation for data objects can be both automatic (in which storage is released when control exits from the program unit) or static (in which storage space is saved throughout the entire execution of the program). Automatic allocation uses storage efficiently but makes certain data-usage errors possible.

The DEFINE construct associates a name with a text string such that whenever that name is referenced, the text string replaces it. DEFINE statements can be nested and can be redefined based upon scope. Thus, while the capability is extremely useful, it adds another dimension of complexity to JOVIAL programs.

Unfortunately for advocates of structured programming, the control statements in JOVIAL J73 are not confined to the "structured programming" constructs of sequential flow, IF-THEN-ELSE, and WHILE-loops. The language does at least have these constructs, so that programmers can write structured code if they desire. However, unstructured

statements as GOTO, FALLTHRU, EXIT, and ABORT are also permitted. The GOTO statement allows transfer from the outside of an IF or CASE construct into the body of the IF or CASE. GOTO statements can also be directed to labels that are external to a program unit or module, if the label is passed as a parameter. The FALLTHRU statement allows control to pass from one CASE alternative to another without making the test normally required at each CASE option. The EXIT statement allows escape out of an immediately-enclosing loop. The ABORT statement provides transfer of control to the label specified in the most recently executed, currently active procedure having an ABORT phrase. Thus, control transfer is not defined until execution time.

The unstructured control statements provide flexibility and execution-time efficiency; but at the same time they increase the chance of committing errors and make the program more difficult to understand. Since 60% of the total cost of software is generally attributed to maintenance, source code scrutability is important.

J73AVS will provide extensive static and data-flow analysis to detect and report possible errors regarding control transfers, data contention due to static allocation, uninitialized variables, structurally unreachable code, potential infinite loops, etc. Program analysis reports can be generated on command by the user to describe such detailed information as DEFINE usage, label references, symbol properties, and global data.

2.2 CHARACTERISTICS OF APPLICATION PROGRAMS

The programs that will be implemented in JOVIAL J73 will be of similar nature to those written in the separate JOVIAL dialects: J3, J3B, and J73/I. Applications will be for navigation, information management, flight controls, communications, etc. The software characteristics of the applications are varied. For example, flight control software has the following characteristics:

- synchronization
- distributed processing
- structurally simple control statements
- simple data types
- real-time processing

On the other hand, applications such as command and control systems have very different characteristics such as:

- batch and interactive modes
- complex data structures
- complex control structures
- large, monolithic modules
- non-real-time processing

Avionics applications are often destined for small on-board computers. For those computers not having a JOVIAL J73 or J73-subset compiler, the programs are developed on a host machine and cross-compiled to the target machine. As is described in Appendix B, there are no software checkout tools available on these small computers, so an AVS operating on the host computer must supply as much assistance as possible to detect errors in program performance and assure some level of testing thoroughness before the program is cross-compiled.

Command and control systems, on the other hand, tend to be very large (several hundred thousand lines of code). They also tend to evolve as needs change. Therefore, not only is testing a major problem, but also code modification and retesting only what is necessary are difficult tasks. In the face of these problems, one of the most valuable assets of any software support tool is the ability to automatically produce concise but helpful program documentation.

2.5 TESTING MEASURES

The problem of determining when a program is error-free is a long way from being solved. However, there are tools available which provide a beginning toward measuring the thoroughness of testing. Rather than wait for a solution to the whole problem, Government and industry should be encouraged to take advantage of these testing measures early in the development of software. The following testing techniques can be used as testing measures since they provide quantitative measures of violations and other reported phenomena (such as statement, branch, or path execution coverage). Furthermore, they are reliable in the sense of always producing the same result (not relying on interpretation):

1. Static analysis to detect coding errors or illegal programming practices.
2. Assertions to specify legal or allowable performance.
3. Statement, branch, or specified path coverage to measure levels of execution.

Software verification without computer-aided testing is extremely expensive. It would be in the spirit of standardization to improve reliability that the Air Force should reassess the testing of computer programs, as described in Air Force Regulation 800-14, to require the use of AVS tools in testing.

3 STUDY OF AUTOMATED TOOLS AND TECHNIQUES

This section discusses the general problem of software testing, describes existing methods and procedures for software verification, provides a chart showing the main characteristics of currently operational AVS tools, and analyzes techniques given in the literature which influenced the design of the JOVIAL J73 tool, J73AVS.

3.1 GENERAL BACKGROUND

3.1.1 Software Verification

Software system verification is a critical problem recognized by developers, customers, and software researchers. The problem is exceedingly complex for large systems. Software verification is a process which analyzes requirements, specifications, and implementation. In addition to determining or proving consistency between each phase of the process, verification includes the problems of determining the reliability, validity, and completeness of the testing phase.

Since verification is such a monumental problem, the approach to improving the situation has been to partition the total process. Requirements, specification, and design languages have been developed to address the early stages of software development, although they have not yet reached a level of widespread acceptance. Compilers and static analyzers attempt to verify semantic and other consistencies within the implemented software. Dynamic and symbolic execution analyzers address software testing more from a functional approach. Test data generation assists with deriving complete test cases from both structural and functional viewpoints. Proof-of-correctness techniques attempt to validate software in a formal way. Even though the partitioning approach has provided considerable progress in the state of software certification, each partition nevertheless has not achieved a high level of maturity or acceptability.

3.1.2 Software Testing

The primary aim of testing is to demonstrate that a system has acceptable performance in terms of its specification. Experience has shown that the software's behavior must be considered over a broader space than the specified functions if testing is to identify errors.

In Fig. 3.1, the universe of software behavior is partitioned in two ways: the specified and unspecified, and the acceptable and unacceptable. Experience with software development tells us that all four of these forms of behavior will exist when software is declared ready for testing, and all four will continue to exist after testing is over, primarily because the testing process is usually confined to examining expected points in the vector space of the input.

In a typical software testing activity, the testing group is attempting to map these regions by probing with single-point test cases. Their success depends on the total resources devoted to exploring the universe of behavior, and on the effectiveness with which they apply those resources in terms of selecting the "best" points for testing. Effectiveness can be improved by the use of a well-designed testing program supported by automated tools.

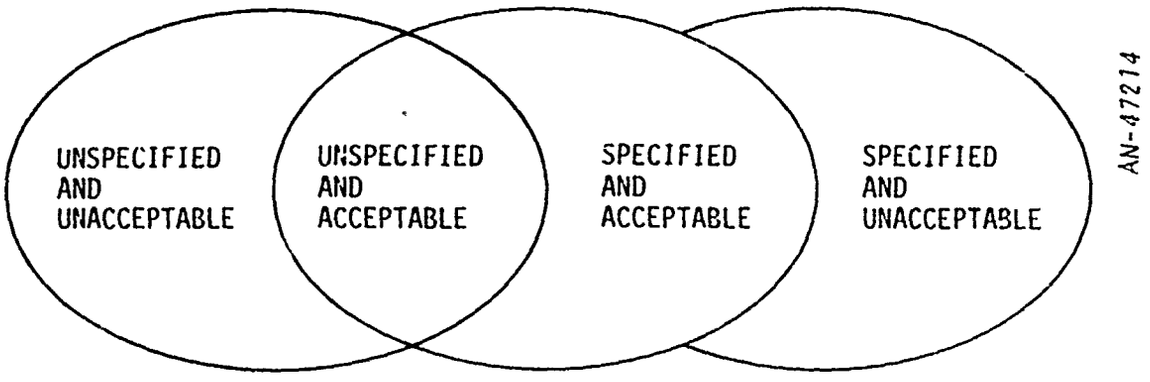


Figure 3.1. Universe of Software Behavior

Software Errors

Since the goal of testing is the detection of errors, we must know something about the characteristics of software errors. Until recently, there was very little data on the types and causes of errors in software systems. Recent studies, however, form a basis of data from which we can state general characteristics of software errors.[1-5] According to these studies:

1. Most errors occur in program logic or in data access, not in computation.

2. Approximately half of all errors are due to errors in specification, and the other half are programming errors as such.

3. Programs do not usually fail catastrophically, but rather errors degrade the program's performance.

4. The scope of errors is usually limited to the one module containing the error.

---

[1] M. J. Fries, Software Error Data Acquisition, Boeing Aerospace Company RADC-TR-77-130, Seattle, Washington, April 1977.(A039916)

[2] A. B. Endres, "An Analysis of Errors and Their Causes in System Programs," IEEE Transactions on Software Engineering, Vol. SE-1, No. 2 (June 1975) p. 140-149.

[3] T. A. Thaye t al., Software Reliability Study, TRW Defense and Space Systems Grou 76-2266.1.9-5, Redondo Beach, California, August 1976.

[4] R. W. Motley and W. D. Brooks, Statistical Prediction of Programming Errors, IBM Corp., Federal Systems Division, RADC-TR-77-175, Arlington, Virginia, May 1977.(A041106)

[5] J. A. Dana and J. D. Blizzard, Verification and Validation for Terminal Defense Program Software: The Development of a Software Error Theory to Classify and Detect Software Errors, Logicon HR 74012, San Pedro, California, May 31, 1974.

These are only broad generalizations that one must be careful in using; there appear to be many confounding factors. For example, the choice of categories for grouping errors can bias the results. Programs written in high-level languages have different types of errors than programs written in assembly language. However, the observation that a majority of programming errors are due to improper sequencing implies that a large amount of the testing effort should be aimed at discovering and correcting these types of errors.

Sequencing in a program is established by the control statements of the program (referred to as the program's control structure). Therefore, it seems natural to base the generation of test cases and test data on techniques which analyze the program's control structure. Several studies and tool developments have pursued this approach, with most of the efforts being applied to test data generation.

Functional Testing

The basic requirement of any system is that it perform its intended function. Functional testing is the means by which the actual behavior is identified; the consequences of this behavior must be related to the intended function through criteria of acceptance derived from the specification. (We ignore in this discussion the frequent occurrence that the specification as interpreted does not represent the intent of the designers). When testing resources are limited, they are applied to testing presumably representative instances of the various functional modes of the system. With more testing resources, functional test cases are usually expanded in an ad hoc manner in an attempt to exercise more of the alternatives that are recognized by the software.

Automation of functional testing usually takes the form of providing a means to step through variations of a basic test case.

Other candidates for automated assistance to functional testing are:

1. Analysis of special representations of input space to assist in the selection of functional test cases
2. Static analysis tools that recognize assertions concerning functional behavior and check for consistency with the code
3. Automated conversion of functional assertions to executable code for execution-time checking against actual results
4. Classification and storage of input data, with mechanisms for generating specific cases
5. Classification and storage of test results, with mechanisms for comparing test results between cases
6. Modification of the input data to map performance boundaries

In addition to the basic purpose of functional tests as a means of demonstrating compliance with acceptance criteria, these tests define the point of departure for extensions to structurally derived tests, described in the next section.

Structure-Based Testing

As Fig. 3.1 suggested, it is the nature of computer-controlled systems that they often display modes of behavior that are not explicitly identified in the specification. The unspecified behavior may result from many different causes, ranging from simple blunders in programming to carefully designed logic that implements an erroneous interpretation of the specification. Often unspecified behavior results when the specification makes no provision for a particular input condition and it is misinterpreted. These unspecified behaviors usually go untested by functional testing.

Structure-based testing is a means of deriving test cases directly from the software with the intent of identifying program paths that are not tested by functional tests, and deriving test data that will cause those paths to be executed. Several test tools now exist which support structure-based testing by detecting which program segments have been executed by a particular test case. The general approach used with such tools is described below.

A graph model of a program module is developed which comprises an input node, an output node, and a set of nodes which represent all the branch points in the module. The nodes are connected by links which correspond to all the straight-line code executed in the program between branch points: the "branches," "logical segments" or "decision-to-decision paths."

Once the graph model is derived, data collection points are automatically inserted in the links to record which links are exercised by a particular test. Then the results of a set of tests are examined to decide how testing of unexercised code should proceed. Most efforts toward further automation of this process have relied on automating a simple rule for test case selection (such as finding a test that reaches a single unexercised target path), and then generating test data for that case. Several tools have implemented approaches to this type of automation (see Sec. 3.2).

3.1.3 Graph Model Theory[1]

This section describes the foundation of graph model theory. This foundation is used as the basis for implementing data flow analysis (a static testing procedure), execution coverage analysis (a dynamic testing procedure), and some automatic test data generation techniques.

---

1 J. P. Benson, et al., Software Verification: A State-of-the-Art Report, General Research Corporation CR-1-638, March 1975.

The use of directed graphs to represent programs is a natural outgrowth of the flow charting practice. There are, however, major differences between a graph and a flow chart: When going from a flow chart to a graph model, some information about the program is unavoidably suppressed. In a graph, attention is drawn to the fundamental control structure of the program (the "paths" and "loops" in the procedure) and not necessarily the calculation being performed.

Program graphs are generally represented in one of two ways. The graph may be described in terms of basic blocks, where a basic block is a linear sequence of program instructions having one entry point (the first instruction executed) and one exit point (the last instruction executed). For a JOVIAL program S consisting of statements $S_1, S_2, \ldots, S_n$, a basic block b is a contiguous subset of the statements of $S[S_i, S_{i+1}, \ldots, S_{i+k}; k > 0]$ having the property that no statement of b, except perhaps S_i, is the destination of any transfer-of-control statement anywhere in S. Alternatively, the graph may be described in terms of branches (or decision-to-decision paths, DD-paths), where a branch is the ordered sequence of statements the program performs as a result of the outcome of a decision up until the evaluation of the predicate in the next decision statement encountered. Figure 3.2 illustrates this definition.

Depending on whether the program graph is described in terms of basic blocks or branches, its nodes and edges have different significance. When basic blocks are used, the blocks are graphed as the nodes, and the transfers of control as the edges, of the graph. The reason is as follows: basic blocks must be physically contiguous statements in a program. They begin on a branching [e.g., GOTO <label>, IF(<condition>)] or labeled statement, and they end on the statement immediately preceding the next branching or labeled statement. Using basic-block terminology, a path through a graph is described as a sequence of nodes.

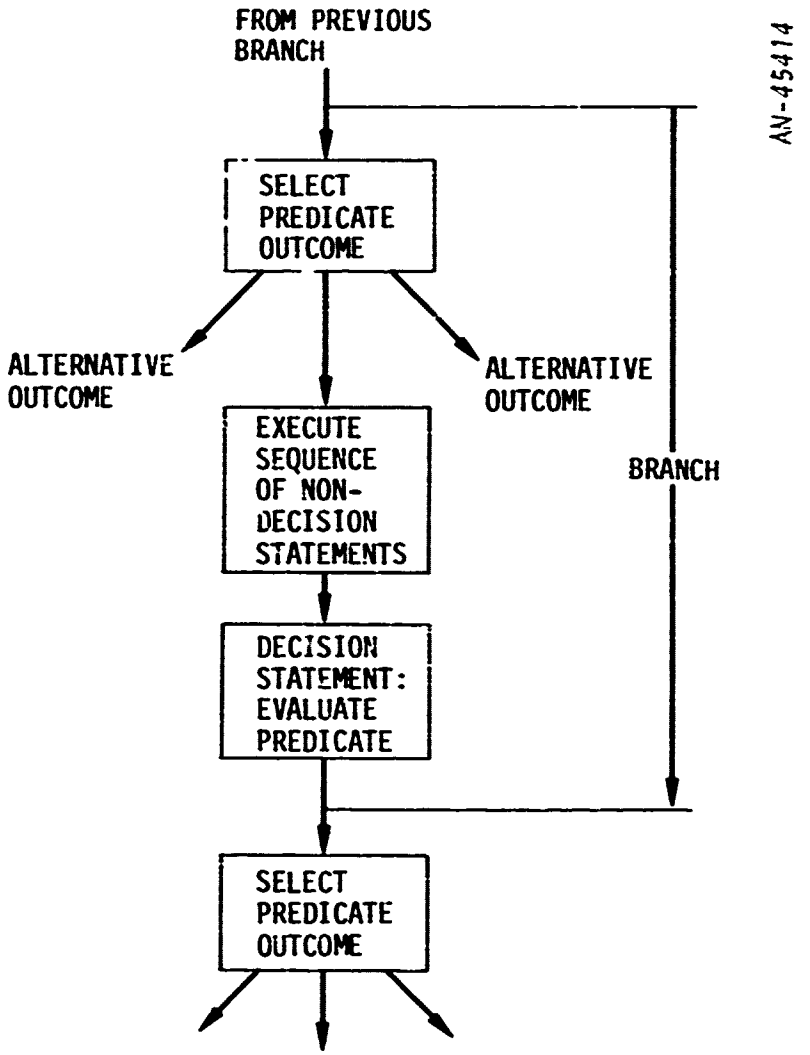


Figure 3.2. Diagram of a Branch

Alternatively, a graph may be described in terms of branches, with the branches as the edges, and the decision statements [e.g., IF <condition>] as the nodes. A branch may include one or more basic blocks that are contiguous in terms of execution. For example, a branch may include an unconditional GOTO statement and the sequential statements that follow its target (labeled statement). Using branch terminology, a path through a graph is described as a sequence of edges.

The following sections describe various techniques which identify processing flows from the graph model of the program.

Depth-First Search

Depth-first search techniques have been applied to a wide variety of practical problems which can be modeled as graphs. Tarjan[1] describes algorithms for implementing the depth-first search, and points out that the algorithms are linearly related to the number of nodes and edges in terms of computation time and storage space. Depth-first search techniques can be used to identify a "spanning tree" for a graph; that is, a subgraph which is a tree and which contains all the nodes of the graph.[1] Algorithms for traversing trees and visiting nodes of a tree can then be applied to the spanning tree. Osterweil and Fosdick[2] have implemented a system which performs data flow analysis using depth-first search techniques. By analyzing a system of FORTRAN modules from the bottom of the calling tree up, the system classifies input/output variables at module interface boundaries. Depth-first search techniques are applied to each module's program graph to determine the input/output classification (i.e., set or used) for all common variables and arguments along all possible paths through the module. Several types of data usage errors can be found while performing this analysis.

---

[1] R. Tarjan, "Depth-First Search and Linear Graph Algorithms," SIAM Jour. Computation, Vol. 1, No. 2, June 1972.

[2] L. Osterweil and L. D. Fosdick, Automated Input/Output Variable Classification as an Aid to Validation of FORTRAN Programs, Dept. of Computer Science, University of Colorado, Boulder, Colorado, CU-CS-037-74, January 1974.

Strongly Connected Components

Tarjan[1] presents an algorithm using depth-first search techniques which identifies "strongly connected" components of a directed graph, in computational time and storage space linearly related to the number of nodes and edges in the graph. A strongly connected component of a program graph identifies an iteration structure. Ramamoorthy[2] describes a procedure similar to this which is to be implemented in an Automated Evaluation Validation System (AEVS). By conceptually replacing strongly connected subgraphs with a subroutine call and a subroutine which contains the iteration structure, and then applying the same procedure to the program graphs of the resulting subroutines, it is possible to abstract an internal calling tree from a single program graph. Ramamoorthy suggests that this technique will be especially useful for large modules with complex iteration structures. The result of abstracting the internal calling tree is that validation analysis can be applied to small non-iterative subgraphs (conceptual subroutines) of the original program graph. The problem of relating this submodule analysis back to the original module still remains unsolved.

Schemes

Sullivan[3] presents a different approach for abstracting a conceptual internal calling tree from the program graph of a module. He refers to a program graph as a scheme. A subscheme is a subgraph of the program graph which has the property that it is a one-entry/one-exit structure. An elementary subscheme is essentially a basic block or DD-path. The decomposition of a scheme by successive partitioning of

---

1 Tarjan, op. cit.

2 C. V. Ramamoorthy, R. C. Cheng, and K. H. Kim, Reliability and Integrity of Large Computer Programs, University of California, Berkeley, ERC-M430, 12 March 1974.

3 J. E. Sullivan, Measuring the Complexity of Computer Software, MITRE MTR-2648, 25 June 1973.

its proper subschemes into further subschemes can be carried out until all subschemes are elementary. The partitioning process creates a conceptual internal calling tree (in which all possible submodules are identified). Sullivan has applied this representation of program structure to the problem of measuring the complexity of computer software.

Intervals

Compiler optimization techniques[1] have fruitfully employed another approach to graphical analysis called interval analysis. Interval analysis is similar to the techniques of identifying strongly connected subgraphs and one-entry/one-exit subgraphs. An interval is a one-entry subgraph which may have one or more exits. Hecht and Ullman[2] describe an algorithm for identifying intervals. A conceptual internal calling tree can be abstracted from program graphs using this algorithm.

Level-i Paths

A technique for identifying program flows explicitly is described by Miller.[3] The manner in which the branches (or DD-paths) described previously can be combined in potentially legal ways in normal program execution is described by objects called "level-i paths." A level-i path is a sequence of DD-paths which lie on the i*th* iteration level within the program, i = 0, 1, 2,.... Because there can be an extremely large number of distinct level-i paths in a program, it is important to consider, instead, classes of level-i paths which lead from the same

---

[1] F. E. Allen, "Control Flow Analysis," SIGPLAN Notes, July 1970.

[2] M. S. Hecht and J. D. Ullman, "Flow Graph Reducibility," SIAM Jour. Computation, Vol. 1, No. 2, June 1972.

[3] E. F. Miller, Jr., A Hierarchy of Program Testing Measures, General Research Corporation, Program Validation Project, February 1974.

nodes and involve the same kind and manner of iteration. Thus, certain forms of parallelism of DD-paths along level-i paths are removed as a means to reduce the combinatoric size of level-i path classes. The result of this reduction is to capture the essentially different program flows in terms of a "principal level-i path" within each level-i path class.

For example, Fig. 3.3 shows a set of DD-paths which corresponds to a program; each DD-path is labeled with a letter. For this particular program graph, the following level-i paths and path classes result:

1. Level-0 path: ab
2. Level-0 path class: $\{cd_ie\}_{i=1}^{m}$
3. Level-1 path: fgh
4. Level-2 path class: $\{k_i\}_{i=1}^{n}$

The level-0 paths represent flow from the input to the output (from the entry to the exit) without iteration; the level-i paths represent ith level iteration "over" constituent level-i paths. DD-paths d_i and k_i represent instances of path parallelism.

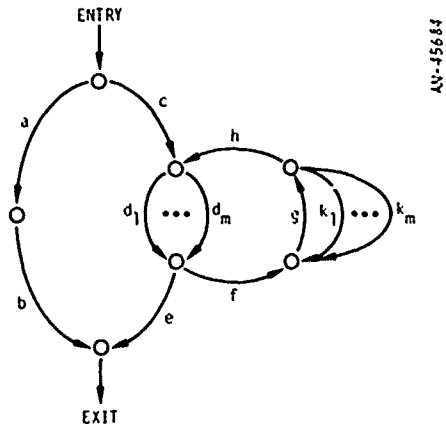


Figure 3.3. Sample Set of Level-i Paths

3.1.4 Static Program Analysis

Enhancing the diagnostics reported, and providing information not usually furnished, by a typical compiler leads to a series of software quality enhancement methods which can be categorized as static analysis. These methods scan the source text of a program for errors in syntax and semantics which can be detected without running the program on a computer, and provide consistency checking and documentation about the definition, reference, and communication of data within the program, Some examples of the supplementary information and error checking are:

Documentation

Cross Reference. A symbol cross reference for each program including symbol type, definition, and use.

Local Storage Identification. All variables used as local storage by a program are identified by their type and use.

Communication Space Analysis. All variables which participate in the communication to other programs (parameters, global variables) are identified according to their use and type.

Parameter Analysis. Variables used as formal parameters to the program are identified and listed along with their use and type.

Identification of Control Variables. Variables which affect the flow of control in a program and where they are referenced are identified.

Consistency Checking

Array Subscript Check. Each subscripted variable reference is checked against the array declaration.

Expression Mode Check. A check is made for expressions whose arithmetic mode changes when they are assigned to a variable.

Local Memory Check. All variables which have the possibility of remaining defined over successive invocations of the program are

identified and their use specified (i.e., JOVIAL static variables).

Argument Check. Formal and actual parameters are checked for inconsistencies in type, mode, number, dimensionality, and use.

In general, static analyzers are most useful in providing the programmer with information which will help debug programs more quickly. They do this by identifying programming constructs which may be legal but risky and providing global, organized information about the identifiers used in this program.

3.1.5 Dynamic Program Analysis

Two basic types of dynamic program analysis are described in this section: analysis of statement-level behavior and analysis of execution coverage. These two techniques are well-known, general-purpose testing aids.

Statement-level Analysis

In statement-level dynamic analysis all program statements are instrumented in order to obtain detailed information concerning the program's internal behavior. This technique produces more detailed and more source-program-oriented information than such earlier techniques as hardware monitoring, software monitoring ("snapshots"), and simulation techniques. Typically, a statement-level preprocessor automatically augments each source program statement with other constructed statements or invocations of run-time subroutines which take measurements while the program is running. These measurements usually include the values of selected program variables and the number and types of branches taken. Examples of the type of data which might be gathered for a JOVIAL J73 program include:

1. An execution count for all statements; i.e., the number of times each statement was encountered during execution

2. For assignment statements, the initial, final, minimum, and maximum values of the computed variable

3. For IF statements, a count of the number of times the IF-expression was true and the final value of the IF-expression

4. Branch counts on each CASE statement, along with the initial and final values of the case selector

5. The initial and final values of the loop-control of FOR statements

6. The number of times a FOR loop was exited "normally," i.e., after doing the specified maximum number of iterations

When the program terminates, summary reports are printed which show the ranges of the program's intermediate variable values, which branches were taken and with what frequency, and which statements in the program were not executed.

Execution Coverage Analysis

This technique attempts to gather information on the run-time sequencing of a program and the flow of control among the various programs comprising a programming system. This sequencing information can be represented at various levels of detail. At the lowest level it may be a trace of the statements executed by a program when run with a particular testcase, or the sequence of branches executed by the program. At a higher level, the actual program flows traversed by the program may be collected or, at a still higher level, the dynamic calling sequence of procedures and subroutines in a programming system may be monitored.

The technique for implementing program flow analysis is the same as that for statement-level analysis, that is, software probes are placed in the programs to be monitored at the level at which the monitoring information is to be gathered. The instrumentation statements are simply invocations of run-time auditing procedures which record which procedure and which control sequence or statement is being executed at the time of the monitoring. A post-processor can then reproduce the dynamic flow of control through a single program or a group of programs at whatever level is desired. This information is useful in determining which control flows and procedures were exercised by which test cases as a guide to what testing remains to be done.

3.1.6 Automatic Test Case Generation

Howden[1] describes a methodology for identifying some of the test cases for a program automatically. His method first partitions the flow of control in a program into standard classes of paths much in the same way as Miller.[2] Then, descriptions of the path classes by predicates and relations are constructed in the form of a system of inequalities. Howden notes, however, that it may not be possible to derive these descriptions for arbitrary programs containing loops. If these descriptions can be generated, the last phase of the methodology is to solve the system of inequalities and thereby derive input values which will cause the program to execute a particular class of control flow. The report by Howden[1] elaborates on the techniques employed in each phase of the methodology, and discusses problems which arise in implementing these techniques. Phases one and two have been partially implemented for analyzing FORTRAN programs.

---

[1] W. Howden, Methodology for the Automatic Generation of Program Test Data, Dept. of Information and Computer Science TR 41, University of California, Irvine, 15 February 1974.

[2] E. F. Miller, Jr. and R. A. Melton, (General Research Corporation), "Automated Generation of Test Case Data Sets," Proceedings 1975 International Conference on Reliable Software, Los Angeles, 21-23 April 1975.

The SELECT system[1] has been implemented by the Computer Science Group at SRI to process an experimental language which resembles a subset of LISP. This system attempts to generate program test cases automatically from the program's semantic and control structure. In contrast to Howden's approach, SELECT does not initially identify classes of program flow, but rather "executes" the program text symbolically, accumulating information as it goes. When a decision is encountered, SELECT keeps track of all the branches resulting from the decision and tries to remove those branches which cannot be executed due to the outcomes of previous decisions. In this way, impossible paths are eliminated as they arise. Two key features of the SELECT system are the adding of "pseudo" predicates and paths for array references and the ability to append a Boolean function to the program under test which returns true if the program satisfies its specification and false if it does not. SELECT then attempts to derive a test case which makes this function return a false value, thereby giving an input for which the program will fail.

Semi-automatic testcase generation for the purpose of extending testing coverage is discussed by Miller.[2] In this method it is assumed that some testing has been done on the program and the goal is to derive a test case for executing a previously untested segment of code. The first step is to identify a sequence of branches which "reach" the untested code segment. This sequence is identified by a flow analysis algorithm which operates on the program graph model. The sequence of branches corresponds to the sequence of statements which must be

---

[1] R. S. Boyer, B. Elspas, and K. N. Levitt, "SELECT--A System for Testing and Debugging Programs by Symbolic Execution," Proceedings 1975 International Conference on Reliable Software, Los Angeles, California, April 1975.

[2] E. F. Miller, Jr., and R. A. Melton, op. cit.

executed in order to reach the untested code segment. This statement sequence is then "backtracked" (symbolically executed in reverse order) in order to identify particular input conditions which will lead to the execution of the untested code segment.

3.2 EXISTING METHODS AND PROCEDURES

There is a wealth of published information on software verification. No one, we are sure, has personally tried all the various manual and automated techniques to evaluate them first hand. For the most part, software verification is still a strictly-manual process. Tools and techniques exist, but this area of software engineering is in its infancy. Most of the tools and methodologies have severe restrictions or require highly-skilled persons to make their application successful.

Some of the current processes that make up software verification are listed below:

Requirements

Requirements state what a computer system should do from the user's viewpoint. Manual systems exist which decompose systems graphically (SADT from SofTech and AXES from Higher Order Software) and which tag requirements for later keying to design and code (THREADS from Computer Sciences Corporation).

Specification

At least two languages and tools exist for stating detailed specifications (Requirements Specification Language - RSL - from TRW and SPECIAL from SRI). Both provide a rigorous means of stating specifications which can be used to detect inconsistencies. Both require considerable expertise to use and provide maximum benefit when applied to large system developments.

HIPO (Hierarchy plus Input-Process-Output) charts are a manual means of stating software specifications in the context of program structure.

Design

There are many design methodologies based upon decomposition, structure, data relationships, top-down and bottom-up development. There are also systems and languages such as Process Design System (PDS - from the System Development Corporation) and Process Design Language (PDL). PDL is a control-structure keyword recognizer.

Functional and Performance Testing

Manual functional and performance testing is assisted by deriving data from HIPO charts, using simulations, obtaining execution-time intermediate-value printout, and running stress or boundary tests by choosing data sets from the specification. Tool-assisted functional and performance testing can be performed by using executable, logical assertions which report inconsistencies between specified and actual behavior; timing analysis where computer clock times are reported at module entries, exits, or branch points; or adaptive testing (the Adaptive Tester from General Research Corporation) where performance boundaries are determined by automatically perturbing the input space.

Structure-based Testing

This testing concept has been very popular for providing a measure for testing completeness, test data generation, error location, and finding structural anomalies. There are a number of automated tools which perform branch testing (RXVP, JAVS, FAVS, SQLAB, and TAP from GRC, NODAL from TRW, PET from McDonnell Douglas, Test Coverage Analyzer from Boeing) or user-specified sequences of statements (SADAT from Kernforschungszentrum Karlsruhe GmbH). Algorithms are being developed which attempt to partition the impossible goal of testing all control paths in a program. Some of these techniques are (1) identifying strongly-

connected components of a directed graph (Tarjan, Ramamoorthy), (2) partitioning the program graph into subschemes which are single-entry/-single-exit structures (Sullivan), (3) identifying strongly-connected subgraphs which are single-entry/multiple-exit, called intervals (Hecht and Ullman) and (4) partitioning the program graph in terms of its iteration level, called level-i paths (Miller).

Manual structure-based testing can be assisted by deriving decision tables (Goodenough and Gerhart) and choosing input data accordingly.

Structural anomalies such as dead code, potential infinite loops, and infeasible paths can be determined by some current AVS tools (ATDG from TRW, SADAT, JAVS).

Consistency Checking

The most common techniques used to determine the consistency of variables and interfaces are adding assertions to state expected use (SQLAB from GRC, ACES from UC Berkeley); employing static analysis (AMPIC from Logicon, DAVE from University of Colorado, FACES from UC Berkeley, RXVP, FAVS and SQLAB from GRC); using data flow analysis to find uninitialized variables and interface inconsistencies (DAVE, RXVP, SQLAB).

Test Data Generation

A great deal of research energy has been expended on developing test data generators. So far, the tools being developed to perform automatic test data generation, such as ATTEST at the University of Massachusetts, are still research oriented and have had to back off from original goals. Other tools such as test harnesses or the Adaptive Tester require input boundaries and invariances between variables to be specified.

For manual test data generation, Howden suggests that input data be chosen to reflect special values for the program. Ostrand and Weyuker suggest deriving data in two phases based upon likely errors for the particular program's function and likely errors for the control structures used in the program. The possible worthwhile approaches to generating test data are too numerous to elaborate here.

Formal Verification

Automated formal verification systems (EFFIGY from IBM, PROGRAM VERIFIER from USC/ISI, SID from the University of Texas at Austin, SQLAB from GRC, SELECT from SRI) take user-supplied assertions (called verification conditions) usually at each branch, and symbolically execute them. The systems attempt to prove each VC as it is symbolically executed. The process involves simplification of inequalities and, in the case of interactive provers, the input of occasional rules to aid simplification. Formal verification is still reserved for small programs. Most of the implemented systems are LISP based.

Program Modification

Tools which utilize a database system and save interface descriptions or other such system-wide information can be helpful to support program modification and maintenance activities. Valuable information for these activities are module interaction reports, detection of global changes, and local updates. Some of the tools that provide this assistance are the Boeing Support Software, SID, JAVS, FAVS, and SQLAB.

Documentation

Automatically-generated reports which provide information about program structure, calling hierarchy, local and global symbol usage, and input and output statement location are very useful during program development, testing, and maintenance. Most AVS tools provide some or all of these reporting capabilities.

3.3 CURRENTLY IMPLEMENTED TEST TOOLS

This section presents a chart of current, operational tools for testing, test case generation, proof of correctness, and coding standards checking. There are numerous other systems in various stages of development, but this chart is restricted to tools that are of substantial value and operate at one or more computer installations.

TABLE 3.1

TEST TOOL SYSTEM INFORMATION

Test Tool System	Developer	Implementation Language	Target Language	Computers	Capabilities of Tool
ACES	U. of Calif. BERKELEY	FORTRAN	CDC/FTN	CDC 6600 IBM 360 UNIVAC 1108	CROSS REFERENCE ASSERTION RANGE CHECK CODING STANDARDS VIOLATIONS (2, 8, 9, 10) DO LOOP VIOLATIONS (4)
ADEPT	[illegible]	[illegible]	FORTRAN/ ASSEMBLY	IBM 370	DO LOOP VIOLATIONS (5) CROSS REFERENCE CODING STANDARDS VIOLATIONS (2, 9, 10) SYMBOLIC EXECUTION PARAMETER VIOLATIONS (3, 4) STRUCTURED TRANSLATION
AIDS	RSS	FORTRAN	FORTRAN	UNIVAC 1110	SYMBOLIC EXECUTION TO DETERMINE: 1. LOGICALLY POSSIBLE PATHS 2. PREDICATES TO REACH EXEC. PATHS
ATLAST	U. of Wash		FORTRAN		SYMBOLIC EXECUTION FOR TEST DATA GENERATION
DAVE	U. of COLORADO	FORTRAN	FORTRAN	CDC 6400 CDC 7600	DO LOOP VIOLATIONS SET/USE VIOLATIONS (1-3) PARAMETER VIOLATIONS (1-3) COMMON BLOCK VIOLATIONS (1,3)
DISSECT	MCDONNELL DOUGLAS	LISP	FORTRAN	PDP-10	SYMBOLIC TESTING
EFFIGY	IBM	PL/I	PL/I	IBM 370	SYMBOLIC EXECUTION THEOREM PROVING
FORTRAN ANALYZER	RSS	FORTRAN	FORTRAN	UNIVAC 1108	STATEMENT TYPE STATISTICS STATEMENT INSTRUMENTATION
FACES	U. OF CALIF. BERKELEY	FORTRAN	FORTRAN	CDC 6400 IBM 360 UNIVAC 1108	CROSS REFERENCE SET/USE VIOLATIONS (1) PARAMETER VIOLATIONS (4) COMMON BLOCK VIOLATIONS (2, 3, 4) DO LOOP VIOLATIONS (1, 3)

TABLE 3.1

TEST TOOL SYSTEM INFORMATION (continued)

FAVS	GENERAL RESEARCH CORP.	DMATRAN OR FORTRAN	DMATRAN OR FORTRAN	CDC 6400 VAX-11/780	CODING STANDARDS VIOLATIONS (2) DO LOOP VIOLATIONS (4) ASSERTION VIOLATIONS (2, 3) SET/USE VIOLATIONS (1, 3) PARAMETER VIOLATIONS (2, 4) BRANCH INSTRUMENTATION AUTOMATED PROGRAM ANALYSIS DOCUMENTATION BRANCH EXECUTION COVERAGE REACHING SET GENERATION STRUCTURING FORTRAN INTO DMATRAN
JAVS	GENERAL RESEARCH CORP.	JOVIAL J3 AND FORTRAN	JOVIAL J3	CDC 6600 HIS 6180	BRANCH AND STATEMENT EXECUTION COVERAGE BRANCH TRACING AUTOMATED PROGRAM ANALYSIS DOCUMENTATION REACHING SET GENERATION ASSERTION VIOLATIONS (3)
MODAL	TRW	FORTRAN	FORTRAN	IBM 360 CDC 6400 UNIVAC 1108	STATEMENT TYPE STATISTICS PATH SEGMENT EXECUTION COVERAGE PATH SEGMENT EXECUTION TRACING
PACE	TRW	FORTRAN	FORTRAN	CDC 6400 IBM 360 UNIVAC 1108	STATEMENT TYPE STATISTICS STATEMENT INSTRUMENTATION
PET	MCDONNELL DOUGLAS	FORTRAN	FORTRAN	CDC 6600/7600 IBM 360/370 HIS 6000 GE 600	STATEMENT TYPE STATISTICS STATEMENT INSTRUMENTATION
PFORT	BELL LABS	PFORT	FORTRAN	IBM BURROUGHS HONEYWELL CDC	CROSS REFERENCE COMMON BLOCK VIOLATIONS (3) PARAMETER VIOLATIONS (1-5)
PROGRAM VERIFIER	USC/ISI	REDUCE	PASCAL	DEC-10	PROGRAM PROVER
QUALIFIER	COMPUTER SOFTWARE ANALYSTS		FORTRAN COBOL JOVIAL ASSEMBLY		STATEMENT TYPE STATISTICS STATEMENT INSTRUMENTATION DATA INSTRUMENTATION

TABLE 3.1

TEST TOOL SYSTEM INFORMATION (continued)

RXVP80	GENERAL RESEARCH CORP.	IFTRAN OR FORTRAN	IFTRAN FORTRAN	CDC 6400/7600 IBM 370 VAX-11/780	AUTOMATED PROGRAM ANALYSIS DOCUMENTATION BRANCH EXECUTION COVERAGE CODING STANDARDS VIOLATIONS (2) LOOP VIOLATIONS (4) ASSERTION VIOLATIONS (2, 3) SET/USE VIOLATIONS (1, 3) PARAMETER VIOLATIONS (2, 4) REACHING SET GENERATION
SADAT	IDT KARLSRUHE, GERMANY	PL/1	FORTRAN	IBM 370	BRANCH EXECUTION COVERAGE SYMBOLIC EXECUTION FOR PATH PREDICATES SYMBOL CLASSIFICATION
SQLAB	GENERAL RESEARCH CORP.	IFTRAN	FORTRAN IFTRAN JOVIAL J3-B PAS L VPASCAL	CDC 6400/7600	DO LOOP VIOLATIONS (4, 5) SET/USE VIOLATIONS (1, 2, 3) CODING STANDARDS VIOLATIONS (2, 9, 10) PARAMETER VIOLATIONS (2, 4) BRANCH EXECUTION COVERAGE AUTOMATED PROGRAM ANALYSIS DOCUMENTATION SYMBOLIC EXECUTION FOR VERIFICATION ASSERTION VIOLATIONS (1, 2, 3) REACHING SET GENERATION
SELECT	SRI	LISP	LISP	DEC-10	SYMBOLIC EXECUTION PROGRAM PROVER
SID	U. OF TEXAS AUSTIN	LISP, REDUCE	GYPSY		INCREMENTAL SOFTWARE DESIGN INTERFACE CHECKING SYMBOLIC EXECUTION FOR VERIFICATION
STANDARDS AUDITOR	COMPUTER SOFTWARE ANALYSTS		FORTRAN COBOL COMPASS	CDC	CODING STANDARDS VIOLATIONS (1-8) COMMON BLOCK VIOLATIONS (3) DO LOOP VIOLATIONS (1)
SURVAYOR	TRW	FORTRAN	FORTRAN		SET/USE VIOLATIONS (1-3) VARIABLE USAGE DOCUMENTATION
TAP	GENERAL RESEARCH CORP.	FORTRAN OR IFTRAN	FORTRAN OR IFTRAN	CDC 6400 IBM 360	BRANCH AND STATEMENT EXECUTION COVERAGE STATEMENT INSTRUMENTATION STATEMENT AND VARIABLE EXECUTION VALUES
TEST COVERAGE ANALYZER	BOEING AEROSPACE	ASSEMBLY	JOVIAL J73/I	DEC	BRANCH EXECUTION COVERAGE

TABLE 3.2
LEGEND FOR CAPABILITIES

CODING STANDARDS VIOLATIONS

1. Statement labels out of order
2. Mixed mode arithmetic
3. Computed GOTO where GOTO variable untested
4. Assigned GOTO
5. Comment card format not standard
6. Statement in inappropriate columns
7. More than 100 statements in routine
8. Undefined labels
9. Unreferenced labels
10. Statement with no predecessor

DO LOOP VIOLATIONS

1. DO loop nests exceed six levels
2. DO loop index used after loop
3. Loop termination data-dependent
4. Uninitialized loop variable
5. No exit from loop

ASSERTION VIOLATIONS

1. Physical units inconsistency
2. Input/Output inconsistency
3. Logical assertion false

SET/USE VIOLATIONS

1. Local variables never set
2. Local variables not set on some path
3. Local variables set but not used later

PARAMETER VIOLATIONS

1. Parameters which are neither set nor used
2. Parameters of different length
3. Parameters which are expressions or functions being set
4. Parameters of different type
5. Actual parameters appearing twice in a list one of which is changed

COMMON BLOCK VIOLATIONS

1. Common variables not set or not used
2. Missing common block declarations
3. Unequal common block lengths
4. Mixed-mode common blocks

4 FUNCTIONAL DESCRIPTION OF J73AVS

This section presents a brief description of the capabilities of J73AVS and describes in what phases of the software life cycle the capabilities should be used. A thorough description is provided in the Functional Description.[1]

Our approach to the design of an AVS for JOVIAL J73 is to provide automated assistance for

- program development
- debugging
- testing
- retesting

The approach excludes

- verification of requirements
- verification of specifications
- automated design aids
- formal program verification (proof of correctness)

The techniques for automating these processes are not developed well enough to be reliable for general-purpose, large software systems.

The specifications for the J73 dialect and compilers include rigorous data-type checking and scope rules. The language allows, however, constructs and control structures which demand caution in their usage (such as recursive and reentrant procedures, jumps into certain control structures, abnormal exits, etc.). Further, the language does not contain a mechanism for specifying expected behavior or reporting user-specified abnormalities (since there is no input/output facility).

---

1 C. Gannon and N. B. Brooks, JOVIAL J73 Automated Verification System Functional Description, General Research Corporation CR-1-947, March 1980.

J73AVS will not duplicate the static consistency checking of the compiler, but, rather, provide the following set of facilities to support program development, debugging, testing, maintenance, and documentation of JOVIAL J73 programs:

1. Logical assertions and timing probes (see ACES, FAVS, JAVS, RXVP80, SQLAB in Table 3.1)

2. Static and data flow analysis (see ACES, AMPIC, DAVE, FACES, FAVS, PFORT, RXVP80, SQLAB, STANDARDS AUDITOR, SURVAYOR)

3. Program structure and characteristic reporting (see ACES, FORTRAN ANALYZER, FACES, FAVS, JAVS, NODAL, PACE, PET, PFORT, QUALIFIER, RXVP80, SADAT, SQLAB, SURVAYOR)

4. Statement performance dynamic analysis (see FORTRAN ANALYZER, PACE, PET, QUALIFIER, TAP)

5. Branch, path, and program unit execution coverage analysis (see FAVS, JAVS, NODAL, RXVP80, SADAT, SQLAB, TAP, TEST COVERAGE ANALYZER)

6. Branch and program unit execution trace analysis (see JAVS, NODAL)

7. Execution timing analysis (see JAVS)

8. Structural retesting assistance (see AMPIC, ATDG, ATTEST, DISSECT, EFFIGY, FAVS, JAVS, RXVP80, SQLAB, SELECT)

9. Test history reporting

J73AVS will support interactive and batch facilities since the various stages of program development through testing and maintenance lend themselves to both modes of operation. The command language will be similar for interactive and batch usage, except that the interactive user will be prompted for information where necessary.

4.1 SUMMARY OF CAPABILITIES

A summary of capabilities is provided as a flow diagram in Fig. 4.1. This diagram describes the primary functions supported by J73AVS as well as the sequence in which they are performed. Figure 4.2 shows the interaction between J73AVS and the user. The user can direct the sequence of analysis activities, using information provided at each stage of processing.

Although J73AVS will exist as a single program, it is best considered as a collection of tools or facilities with which the user interacts. Some of the facilities, such as automated documentation, static error reporting, and instrumentation, are completely automated and require only that the user initiate the tasks by command. Other processes, such as execution-time data collection or retesting assistance, require more information from the user like test data input and test target selection.

J73AVS provides detailed information both statically and dynamically about the program being analyzed. It is the role of the user to direct the processing performed by J73AVS, to analyze the output produced by J73AVS, and to determine subsequent action.

The role of J73AVS in the software development cycle is to provide automated assistance wherever possible during the program development and maintenance, debugging, testing, and retesting phases of the cycle. The user of J73AVS plays an active part in the cycle as shown in Fig. 4.3. This figure partitions the phases of the development cycle and

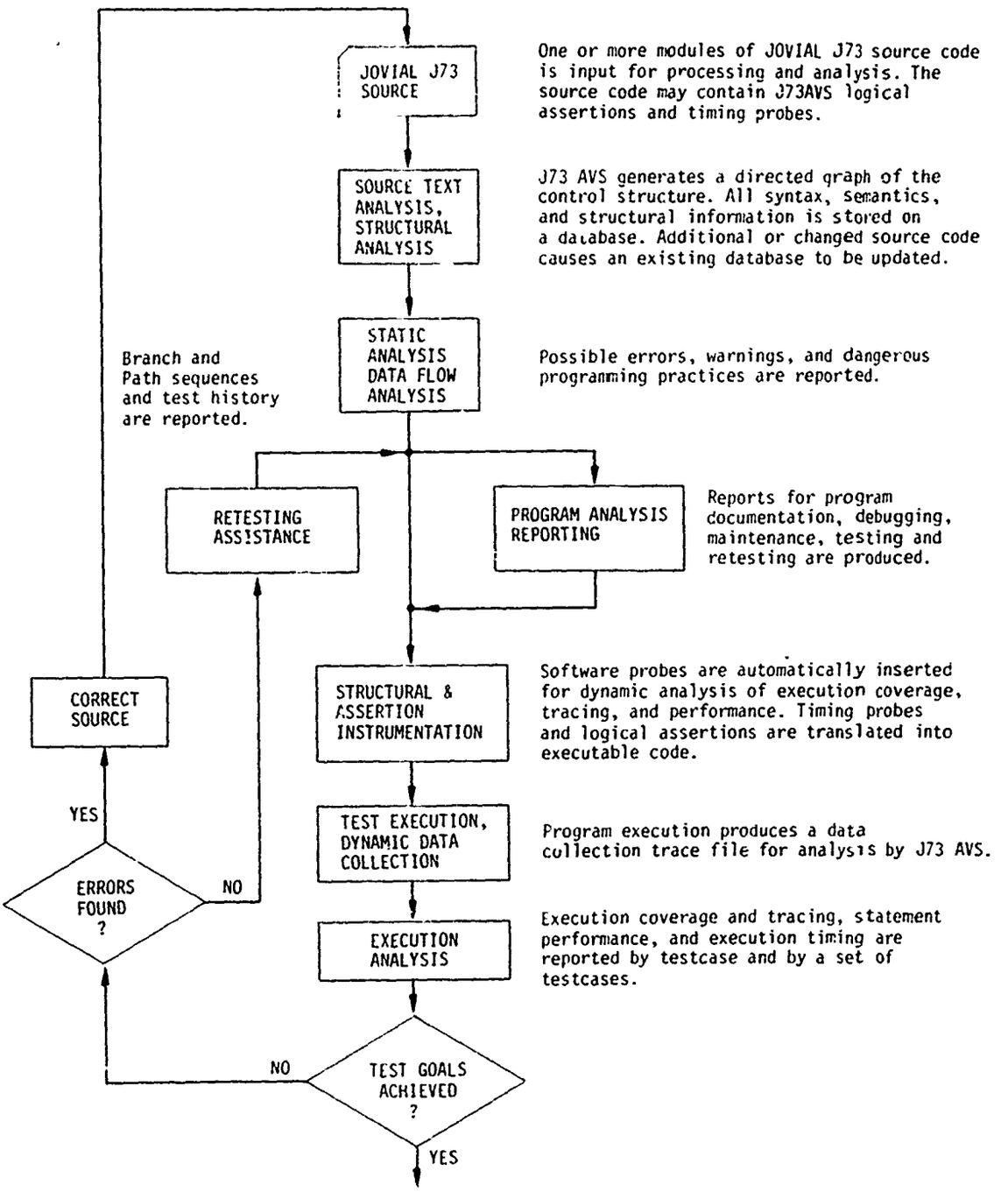


Figure 4.1. Overview of J73AVS

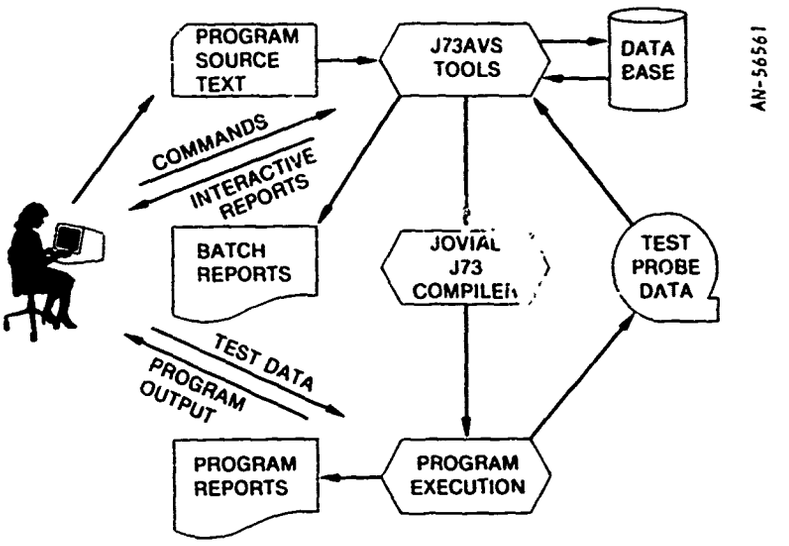


Figure 4.2. J73AVS Interaction with User

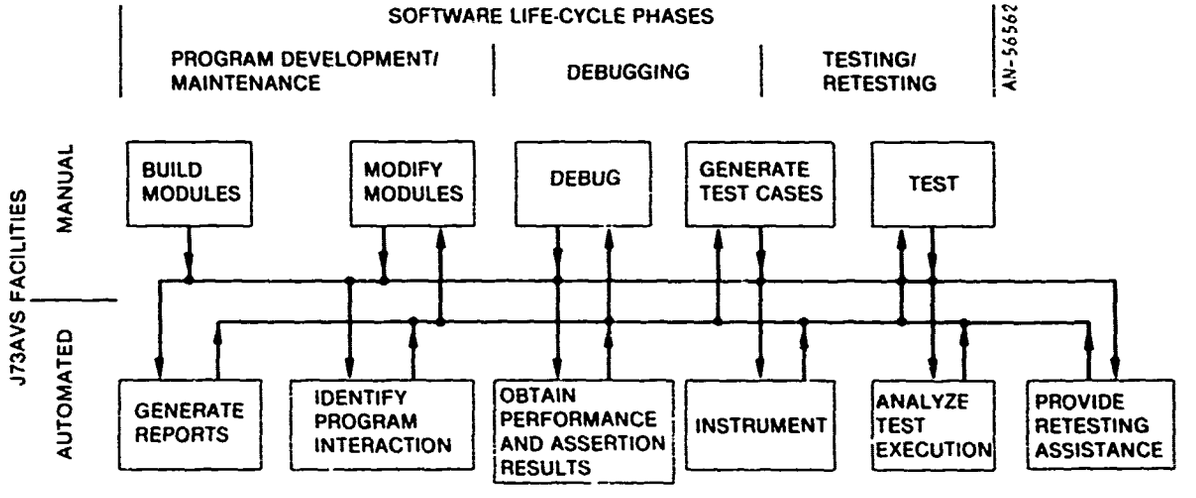


Figure 4.3. Role of J73AVS in the Software Development Cycle

shows the flow between the automated processing of J73AVS and user-supplied input or direction.

Using Fig. 4.3 as a basis, a typical sequence of J73AVS-supported processing can be described as follows:

1. JOVIAL J73 source text is generated and provided to J73AVS as one or more compilable modules.

2. J73AVS produces program analysis reports showing control structure, symbol usage, calling hierarchy, etc., as well as a static analysis report showing errors and dangerous programming practices.

3. Using the reports as a guide, the source modules can be modified or new modules added to the program.

4. J73AVS identifies the interaction of the new or modified modules with the rest of the program; this information, in turn, is used as the basis for modifying other modules.

5. For dynamic debugging, the program is instrumented by J73AVS and executed with an initial test case supplied by the user.

6. J73AVS reports assertion violations, if any, and generates an evaluation of statement and variable performance.

7. Using this evaluation, the user may choose to generate additional test data to pinpoint errors or instrument other modules for additional dynamic debugging.

8. The same procedures of test data generation, instrumentation, and execution are performed for testing but for a different goal: rather than detecting and locating errors, testing aims to demonstrate the absence of errors. Therefore, J73AVS produces execution analysis reports in terms of the thoroughness of execution coverage.

9. The user evaluates execution coverage and other program performance output, along with the program's own execution results and the program specification, to determine if testing is complete.

10. J73AVS provides branch sequence information to retest targets chosen by the user. A test history of execution coverage and assertion violations assists the user in choosing targets for retesting.

Program Development and Maintenance

Executable assertions permit a programmer to specify expected behavior. J73AVS supports the technique of embedding programmer-specified assertions into the code through the use of the ASSERT keyword followed by any legal logical (Boolean) expression. Logical assertions can be used for execution-time exception reporting, stress testing, test data generation filtering, and (left as comments in the source code) stating in-line specifications.

To assist with reliable system development, maintenance, and documentation, J73AVS will provide substantial program analysis reporting on structural hierarchy, symbol usage, invocations, certain J73 constructs, and system characteristics. The user has control over obtaining high- or low-level information through the command language. The types of program analysis reporting include the following:

- indented source listing with control structure identification
- symbol cross reference with set-use information
- compool symbol description
- properties of all or specified symbols
- declaration and reference of labels (statement names)
- declaration and reference of user-defined data types

- declaration and reference of constants
- usage of external reference (REF) and definition (DEF)
- declaration and reference of DEFINE text strings
- description of program units on the database

Debugging

Normal compilation using JOVIAL J73 compilers will detect many syntax and semantic errors. Additional errors such as uninitialized variables, possible infinite loops, unreachable code, certain improper constructs, and dangerous coding practices (like transferring into CASE or IF statements) will be reported by J73AVS. The user can command different levels of static reporting.

Dynamic debugging will be supported by statement execution performance and assertion exception reporting. Statement execution performance provides execution counts of statements, values and ranges of variables in assignments and loops, and the execution behavior of IF statements. This debugging information appears adjacent to the source statements themselves, which assists the task of code correction. The execution of timing probes (inserted by command) can be reported in the debugging performance report at the user's request.

When the program's execution behavior deviates from the acceptable logical behavior specified by the embedded assertions, it will be reported during execution. The user-supplied assertions remain relatively transparent to the program until they are violated; at that time the violation is reported along with the source statement number where the violation occurred.

Testing

When used in conjunction with static checking and statement-level performance analysis, structure-based testing can uncover errors due to

untested branches (where a branch is a control flow outcome due to a decision statement) or improper sequences of branches. J73AVS will provide execution tracing of program units and branches and execution coverage analysis of program units, branches, and sequences of branches (paths). Further, J73AVS will assemble the timing information from program unit tracing and user-supplied timing probes into an execution timing report.

Although an AVS can provide an objective measure of testing thoroughness in terms of statement or branch execution coverage, frequently errors in software are overlooked during testing because only certain sequences of branches are ever executed. Obviously, it is generally impossible to define all paths in programs because of loops. Furthermore, the most likely subset of paths to test can best be identified by a person familiar with the function of the program. The most efficient role of an AVS in this regard is to identify the set of control paths between two statements in a program unit (an invokable unit of code) to which the human tester attaches importance. Of the set of paths identified by the tool, the user can choose those that are to be analyzed for coverage during execution. If the set of paths is too large to enumerate, a descriptive message will be issued and the user allowed to choose another pair of statements for path identification.

Retesting Assistance

Retesting software is performed when analysis shows that prior testing is inadequate (insufficient branch coverage, not all functions demonstrated, etc.) or when program changes have taken place. The proper approach to take in retesting is highly dependent upon the characteristics of the program being tested as well as the measures being used to evaluate testing completeness. A detailed methodology for testing and retesting software for the purpose of improving structural-testing completeness will be given in the User's Manual.

In order to determine the sequences of branches which must be executed in order to reach an untested branch or statement, the user can request that the "reaching set" be computed between two specified statements (or from the program unit's entry). The user can also request a list, in terms of branches, of all control paths between two specified statements. If certain loop structures make this list impossible, subsets of the paths will be identified.

With the control flows identified, the user can backtrack through the program to the input space, using statement execution performance reports, module interaction and invocation reports, and execution coverage information for each testcase to assist in developing new test data. Unfortunately, automatic test data generators which use symbolic execution are not yet developed to the point of being general-purpose, easy to use, or reliable.

The cumulative test coverage history maintained by J73AVS will be useful in attaining testing goals and determining targets for retesting. Program unit and branch coverage information will be saved in a concise way on the database for each test case. The results of subsequent execution runs can be added, providing a cumulative report of all tests. Also saved in the history database table will be any assertion violations that occur. This will provide a mechanism for identifying which input test case caused a violation.

Unfortunately there is no technique that can, in general, echo back to the user what the input for each testcase is. Paragraph 4.1.1.3 of the Statement of Work (PR No. B-9-3278) requested the identification of input test data used for each testcase, but this can be done only in trivial cases such as input on a single file. In complex programs, data are input from a variety of external sources such as databases, subroutine parameters, and files. J73AVS distinguishes separate testcases (as defined by the user) in its post-execution analysis reports but does not print test data input used to drive each testcase.

4.2 J73AVS OPERATION

J73AVS will be implemented to operate in both batch and interactive modes. This versatility provides the user with the ability to customize a debugging and testing strategy to his own software. Depending upon the test object (program being tested) and testing goals, the sequence of J73AVS operations may be varied. Figure 4.1 showed a typical flow of operations, beginning with analysis of previously unanalyzed code and proceeding until some testing goal is realized.

The functions of J73AVS will be driven by user command. The command syntax will be similar for both batch and interactive modes of operation. The command language is made up of specification and operation commands. Specification commands consist of:

BATCH

to notify J73AVS of the mode of operation (the default - no command - is interactive mode) and the text specification commands:

MODULES = name,....	FOR MODULES = name,... (Two or more commands) END FOR
UNITS=name,....	FOR UNITS = name,... (Two or more commands) END FOR
SYSTEM	FOR SYSTEM (Two or more commands) END FOR

J73AVS operation commands control six major functional capabilities: read source text and build database, perform static and data flow analysis, prepare program analysis reports, instrument the source text for dynamic analysis, perform post-execution analysis, and provide

retesting assistance. These commands will have the following syntax (defaults are underlined):

1. Read JOVIAL J73 Source code -
   Command: READ{, CHANGES}

2. Static and data flow analysis -
   Command: STATIC {, LOCAL/GLOBAL, OFF =(ERRORS,
   WARNINGS, MESSAGES, SYMBOLS), SUMMARY/FULL}

3. Program analysis reporting -
   Commands: LIST
   CROSS REF {,MATRIX, SETUSE, NAMES =name,...}
   INVOCATIONS {,MATRIX, TREE, BANDS, SOURCE}
   COMPOOL {,XREF, SOURCE}
   SYMBOLS {,LIST, PROPERTIES, SOURCE, NAMES=name,...}
   LABELS {,LIST, XREF, SOURCE}
   TYPE {,LIST, SOURCE}
   CONSTANTS{,LIST, SOURCE}
   REFDEF {,LIST, SOURCE}
   DEFINE {,LIST, SOURCE}
   DATABASE {, UNITS, DESCRIPTION}

4. Instrumentation for dynamic analysis -
   Commands = INSTRUMENT { ASSERTIONS, STATEMENTS,
   COVERAGE = BRANCH/ENTRY, TRACE=BRANCH/ENTRY}
   TRACESET {UNIT=name, LOCAL/SUBORDINATES,)start smt, stop smt)}
   NEWTEST, UNIT = name, smt.
   ENDTEST, UNIT = name, smt.
   STARTCLOCK, UNIT = name, smt.
   STOPCLOCK, UNIT = name, smt.

5. Post-execution analysis -

Commands = COVERAGE{,ENTRY,BRANCH,STATEMENT,NOTHIT,
HITS=BRANCH/PATHS(path no.,path no.,...)}
TRACE{,ENTRY/BRANCH}
PERFORMANCE {,ASSERTIONS}
TIMING

6. Retesting assistance -

Commands = SETPATH,UNIT=name,BRANCHES=branch1,branch2,
{,branch3,...}* {,RESET}
PATHS,UNIT=name,START=smt.,STOP=smt., LIMIT=number**
BRANCHES.UNIT=name,START=smt.,STOP=smt. {,ITERATIVE}
HISTORY {,RESET}

There are two additional commands: HELP and SAVE. The HELP command is for the interactive user to provide command syntax assistance. The SAVE command is used to save the current contents of the database. The function of each command is briefly described in Table 4.1. A thorough description of each command, along with sample usage and output, is provided in the Functional Description.

Figures 4.4 through 4.12 show input-process-output for the major functional capabilities. Figures 4.4 and 4.5 illustrate the flow of information for commands READ and STATIC. Figures 4.6 through 4.9 illustrate instrumentation and execution of instrumented modules. Figures 4.10 and 4.11 illustrate post-execution analysis, and Fig. 4.12 shows program analysis reporting.

---

* Repetition of branch sequences is denoted by enclosing the branch numbers in parentheses.

** The default number of paths is 50.

Note: All commands can be abbreviated to the first four letters of each keyword.

TABLE 4.1

J73AVS COMMAND LANGUAGE

Command	Parameters	Function
READ	CHANGES	Read HOVAL J73 source. Build database. Identify changed modules on database.
STATIC	LOCAL/GLOBAL, OFF=(ERRORS,WARNINGS, MESSAGES,SYMBOLS), FULL/SUMMARY	Perform static and data flow analysis.
LIST		Produce indented source listing.
CROSS REF	MATRIX,SETUSE,NAMES= name,....	Produce symbol cross reference
INVOCATIONS	MATRIX,TREE,BANDS,SOURCE	Produce reports describing program unit invocation structure.
COMPOOL	XREF,SOURCE	Produce reports describing compool symbol usage.
SYMBOLS	LIST,PROPERTIES,SOURCE, NAMES=name,...	Produce reports describing symbol attributes.
LABELS	LIST,XREF,SOURCE	Produce reports describing statement names (labels).
TYPE	LIST,SOURCE	Produce reports describing user-defined datatypes.
CONSTANTS	LIST,SOURCE	Produce reports describing constant datatypes.
REFDEF	LIST,SOURCE	Produce reports describing instances of REF and DEF specification.
DEFINE	LIST,SOURCE	Produce reports describing instances of DEFINE declaration and reference.
DATABASE	UNITS,DESCRIPTION	Produce reports describing the program units stored in the current database.

TABLE 4.1 Continued

Command	Parameters	Function
INSTRUMENT	ASSERTIONS, STATEMENTS, COVERAGE=BRANCH/ENTRY, TRACE=BRANCH/ENTRY	Inserts software probes into the source code to collect data during execution. Translate assertions into executable code.
TRACESET	UNIT=name, LOCAL/SUBORDINATES, (start smt, stop smt)	Instrument each branch between the specified statements, including branches in subordinate program units.
NEWTEST	UNIT=name, smt.	Insert a testcase boundary at the specified statement.
ENDTEST	UNIT=name, smt.	Insert an end-of-all-testcases probe at the specified statement.
STARTCLOCK	UNIT=name, smt.	Insert a "start" system clock probe at the specified statement.
STOPCLOCK	UNIT=name, smt.	Insert a "stop" system clock probe at the specified statement.
COVERAGE	ENTRY, BRANCH, STATEMENT, NOTHIT, HITS=BRANCH/PATHS(path no,...)	Produce post-execution analysis reports describing statement, branch, or path coverage.
TRACE	ENTRY/BRANCH	Produce a post-execution tracing report for branches or program unit entries and returns.
PERFORMANCE	ASSERTIONS	Produce a post-execution statement performance report, including assertion violations.
TIMING		Produce an execution timing analysis report.
SETPATH	UNIT=name, BRANCHES=..., RESET	Store the specified branch sequences in the database as paths.
PATHS	UNIT=name, START=smt., STOP=smt., LIMIT=no.	Identify the paths between the two specified statements.

TABLE 4.1 Continued

Command	Parameters	Function
BRANCHES	UNIT=name,START=smt., STOP=smt., ITERATIVE	Generate a reaching set of branches between the two specified statements.
HISTORY	RESET	Produce a execution coverage report for all testcases. Reset the database coverage history table.
HELP		Assist with command syntax.
SAVE		Save the current database.
BATCH		Indicate batch mode of operation.
MODULES	name,....	Specify one or more modules for the following command processing.
FOR MODULES	name,...	Specify one or more modules for the following set of commands.
UNITS	name,...	Specify one or more program units for the following command.
FOR UNITS	name,...	Specify one or more program units for the following set of commands.
SYSTEM		Specify all program units in the database for the following command.
FOR SYSTEM		Specify all program units in the database for the following set of commands.
END FOR		Conclude the set of commands.
END		Conclude J73AVS processing.

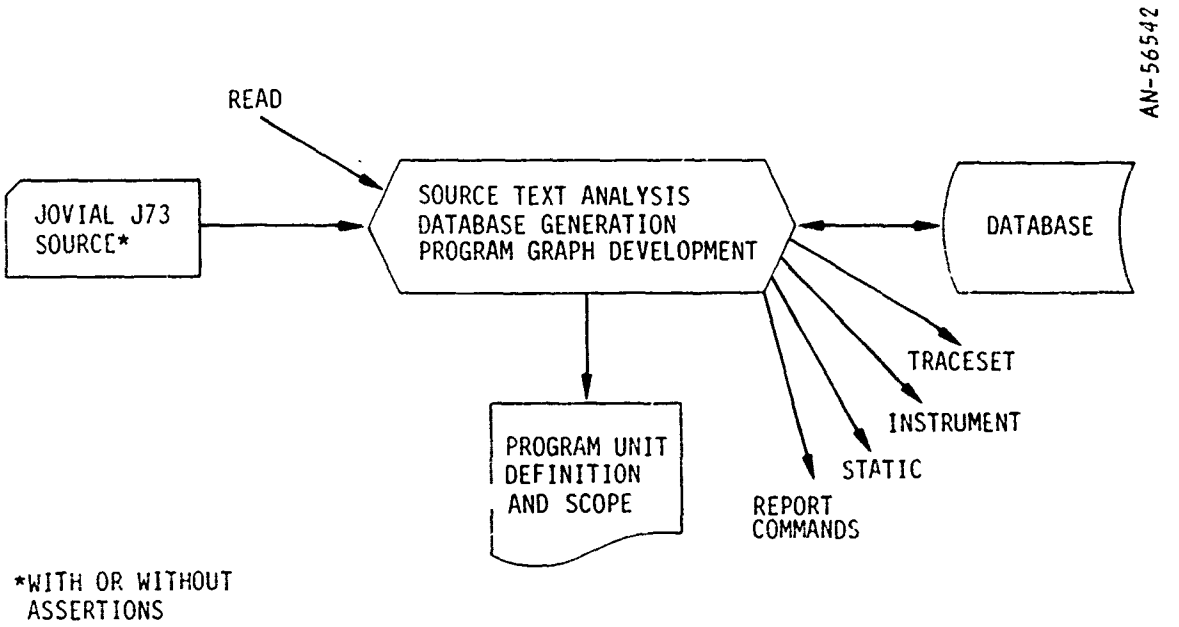


Figure 4.4. Initial Processing

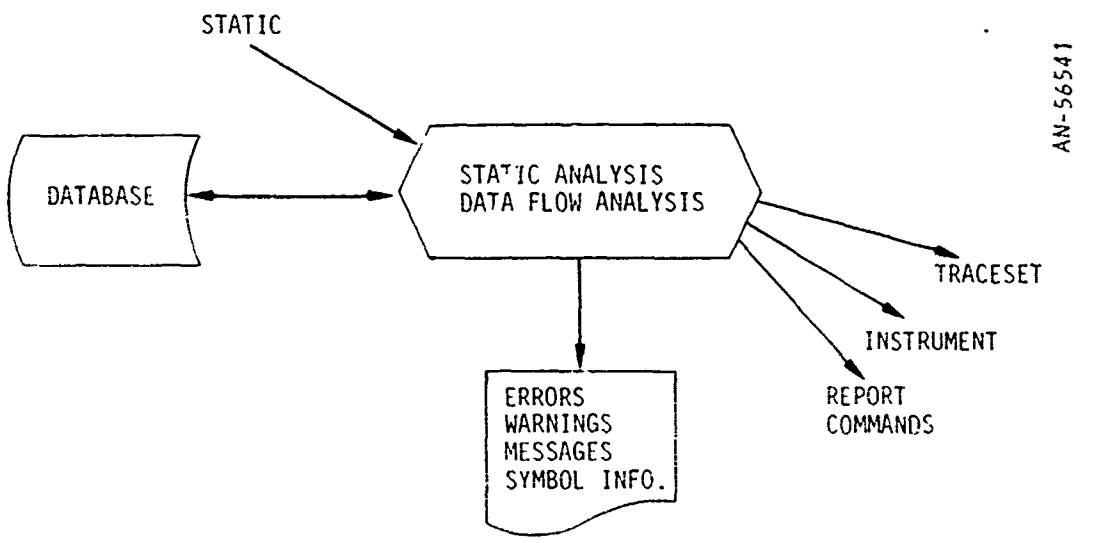


Figure 4.5. Static and Data Flow Analysis

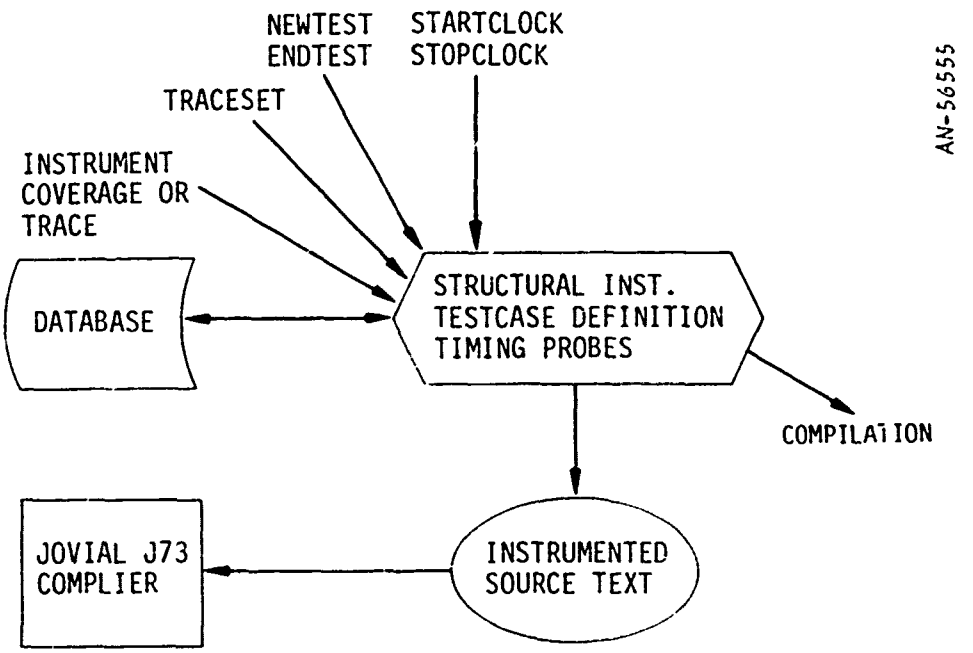


Figure 4.6. Structural Instrumentation

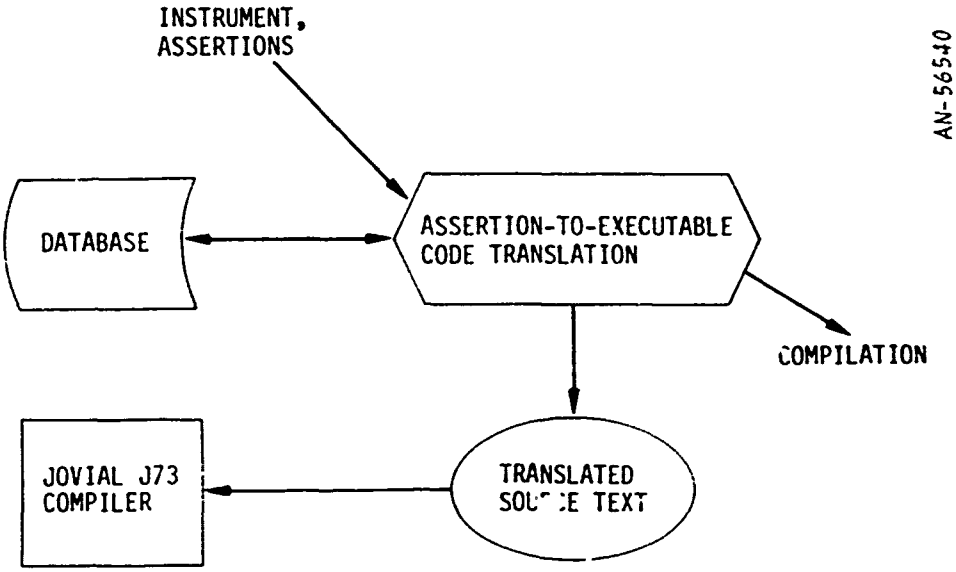


Figure 4.7. Assertion Instrumentation

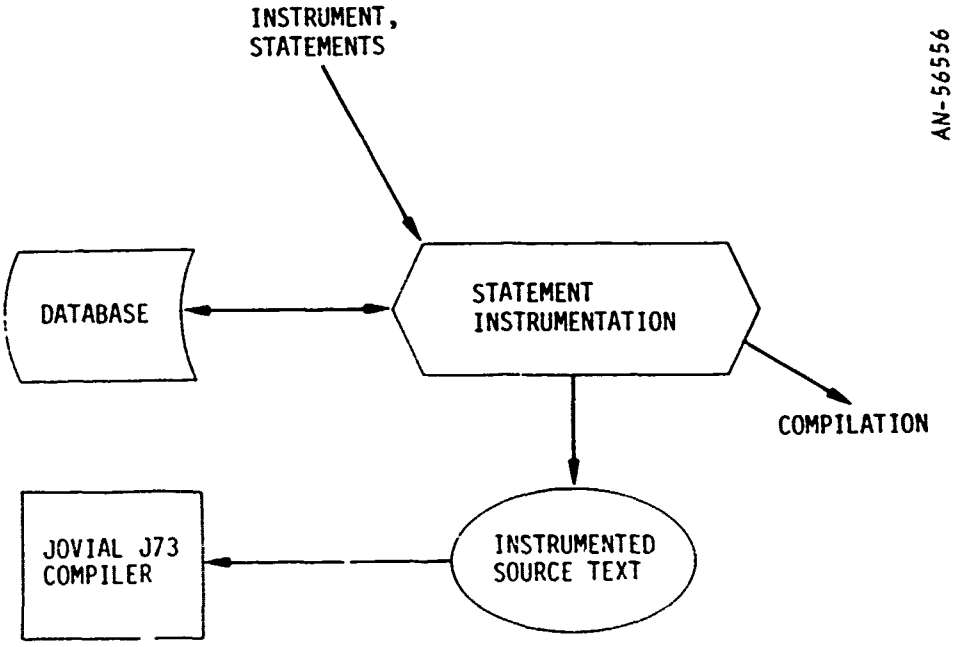


Figure 4.8. Statement Performance Instrumentation

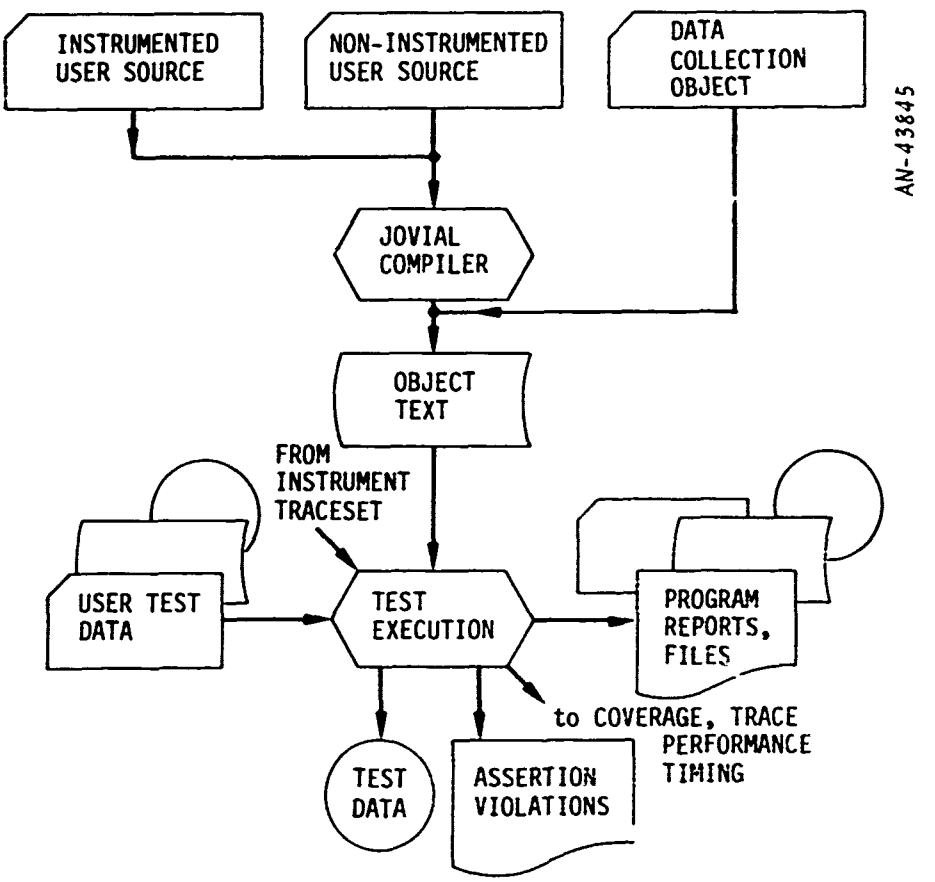


Figure 4.9. Test Execution Processing

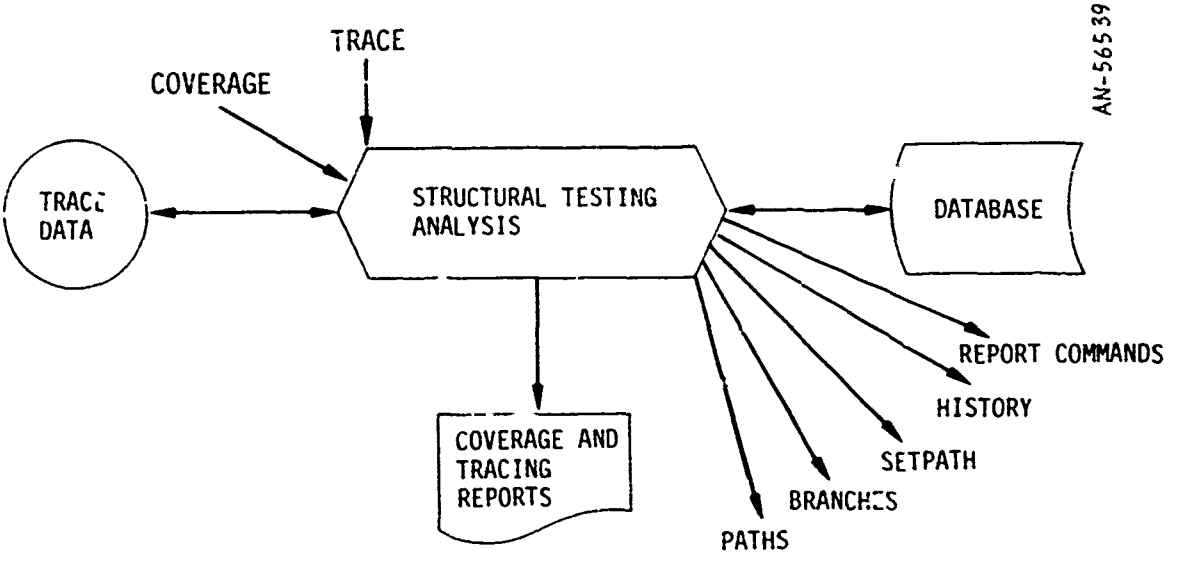


Figure 4.10. Structural Testing Analysis

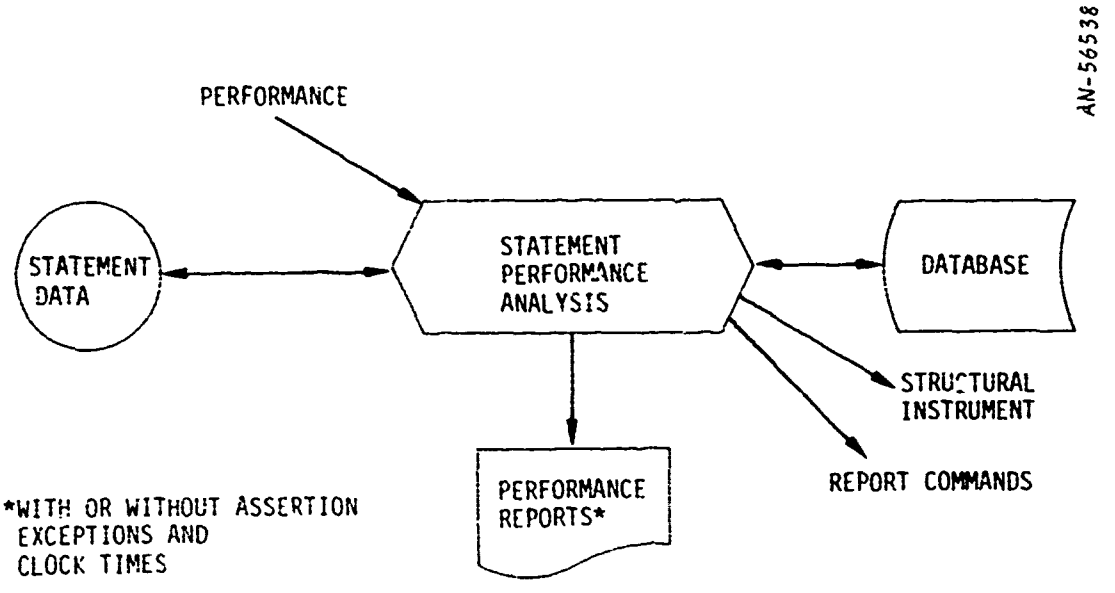


Figure 4.11. Statement Performance Analysis

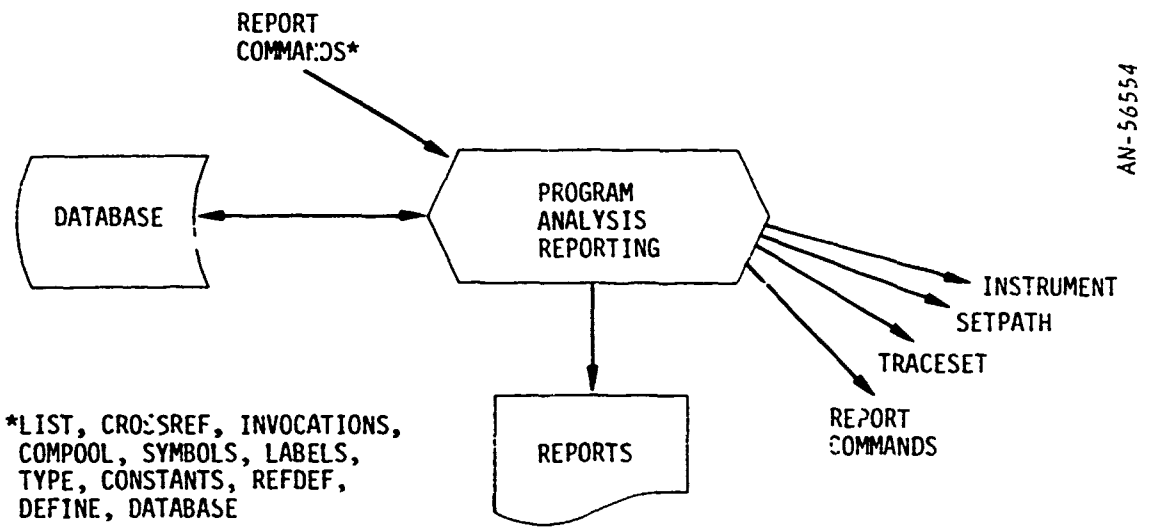


Figure 4.12. Program Analysis Reporting

5 DESIGN OF J73AVS

J73AVS will be made up of a Nucleus and set of independent function processor segments. Each of the segments can correspond to an overlay segment. The Nucleus can make up the core-resident root (or the first level) of the overlaid program, although to minimize storage requirements, some Nucleus routines will be loaded in secondary overlays. Each of the other functions makes up a second-level segment. The following is a brief description of each functional segment:

Command Decoding and Control: Process user input commands, output interactive response, and successively return each command to the overlay controller.

Initialization and Wrapup: Upon run initialization, open files, initiate execution of the storage manager, and set various global data; upon run termination, close files and (for batch mode) produce report index.

JOVIAL J73 Source Text Analysis: Read JOVIAL J73 source and perform lexical scan, token recognition, symbol classification, and structural pointer construction.

Structural Analysis: Build program graph, store branches, and compute single-entry/single-exit reduction history used in data flow analysis.

Supplementary Table Building: Build tables needed for module dependence reporting and cross references.

Program Analysis Reporting: Produce selected reports at user command.

Instrumentation: Insert probes at program unit entries, exits, branches, and statements (depending upon type of instrumentation selected); define new testcase or end of all testcases; expand assertions into executable code.

Structural Testing Analysis: Analyze run-time execution trace file, produce coverage and trace reports, and update test history table.

Statement Performance Analysis: Analyze run-time trace and instrumentation statement descriptions and produce statement performance reports.

Execution Timing Analysis: Analyze run-time execution trace and produce timing report.

Path Generation: Determine the set of paths between specified statements and store paths into database.

Branch Reaching Sets: Generate sets of branches that reach a specified statement.

Test History: Generate a test coverage history report or reset the history table.

Print Services: Print the contents of specified database tables.

Table 5.1 lists the functional processor segments along with the associated user commands which invoke each segment. The Nucleus consists primarily of database management facilities. The segments loaded at a particular time during a run will depend upon the type of processing requested by the user through commands.

TABLE 5.1. J73AVS FUNCTIONAL PROCESSOR SEGMENTS

Command Keyword	Segment No.	Functional Processor Segment
All commands	1	Command Decoding and Control
All commands	2	Initialization and Wrapup
READ	3	JOVIAL J73 Source Text Analysis
READ	4	Structural Analysis
STATIC	5	Static and Data Flow Analysis
INVOCATIONS,CROSSREF	6	Supplementary Table Building
Reports*	7	Program Analysis Reporting
NEWTEST	8	Instrumentation
ENDTEST		
STARTCLOCK		
STOPCLOCK		
INSTRUMENT**		
TRACESET		
COVERAGE	9	Structural Testing Analysis
TRACE		
PERFORMANCE	10	Statement Performance Analysis
TIMING	11	Execution Timing Analysis
SETPATHS,PATHS	12	Path Generation
BRANCHES	13	Branch Reaching Sets
HISTORY	14	Test History
LIST,PRINT***	15	Print Services

*Commands CROSSREF,INVOCATIONS,COMPOOL,SYMBOLS,LABLES,TYPE,CONSTANTS, REFDEF,DEFINE,DATABASE

**Structural instrumentation (parameters COVERAGE and TRACE) and statement performance instrumentation (parameter STATEMENTS) can be sub-overlays.

***Database table print package, primarily for J73AVS development and maintenance and for source listing reports.

The design of J73AVS lends itself to incorporation of changing requirements (such as J73 language revision) and upgrading capabilities. For example, anticipated changes in the JOVIAL J73 language specification (scheduled to be resolved by July 1, 1980) are expected to affect only the syntax analyzer. Upgrades, such as adding a configuration management capability or adding a target machine statement simulator, would be performed by adding new functional segments. The database is designed so that new tables of information can be easily added, and the database manager does not depend upon the type of information stored in the tables.

6 FUTURE EFFORT

There are five techniques for software verification that should be considered for future implementation in J73AVS. The two more important areas are test data generation and instruction-level simulation. Test data generation would be a valuable assistant for all applications to JOVIAL J73. Instruction-level simulation for the purpose of analyzing size, accuracy, and timing for target machines would be beneficial for real-time applications, such as avionics.

Additional, completely-automatable facilities are code auditing, physical units consistency checking, and assertion translation using a precompiler. Detection of certain "dangerous" coding practices is included in the J73AVS static analyzer. It cannot be too strongly stressed that such practices should be retained only for compatibility with existing code; use in new applications should be prohibited except where extreme requirements exist for time and space efficiency. When J73 becomes a familiar language, coding standards should be specified by the JOVIAL User's Group (an Air-Force-sponsored group of interested individuals from industry, Government, and the military) and included in J73AVS. Units consistency checking is already performed in AVS tools such as SQLAB. The addition of this facility to J73AVS would be a small effort.

It has been the practice at GRC to design and develop automated software tools using a top-down, modular approach. Our basic approach is to isolate major functional blocks into software components that have well-defined interfaces. When new or more efficient techniques are developed, they are incorporated into the system as additional or replacement components. Both test data generation and instruction-level simulation can be incorporated into J73AVS as additional functional components.

6.1 TEST DATA GENERATION

In order to implement a test data generation system, the following functional components are required:

1. Syntax analyzer--breaks incoming source text into tokens and stores module, statement, and symbol information into tables for subsequent use.

2. Structural analyzer--generates a directed program graph for each module based on its control structure; saves the control path information in the branch table for later use.

3. Pseudo-path eliminator--this component contains two techniques:
   a. Acting on interactive command from the user, it eliminates sequences of paths from the test case selection process which are logically impossible or "uninteresting" during a particular testing activity.
   b. Using backward symbolic execution, automatically determines and eliminates logically impossible path sequences.

4. Reaching sequence generator--generates reaching sequences according to (1) interactive identification by the user of starting and stopping branches or (2) algorithmic identification of the starting and stopping branches based on execution coverage performance. Also generates individual branch sequences.

5. Reaching sequence constraint generator--builds an expression resulting from the backtracked reaching sequence. Also analyzes individual branch sequences.

6. Constraint simplifier--uses arithmetic, logical, and relational simplification to reduce the path sequence constraint to a set of inequalities. This process should utilize interactive assistance from the user in terms of additional simplification rules.

7. Inequality solver--generates input data for subsequent dynamic execution according to some automated or interactively-supplied heuristics. If the set of inequalities is nonlinear, interactive assistance will be required to determine solutions.

8. Instrumentor--(1) automatically stores software "probes" into the source code so that coverage information can be recorded during dynamic execution, and (2) automatically translates user-supplied assertion statements in the source code into executable statements.

9. Execution analyzer--processes the trace file recorded during execution of the instrumented source code to provide branch and module execution coverage information.

10. Table builder--builds certain tables such as symbol cross reference, module dependence, common symbols, etc. which will be needed for documentation reports and backtracking through the module hierarchy.

11. Report generator--produces a variety of user-specified (through the command language) reports about the characteristics of the test program as a whole or with respect to specified target branches.

The functional components briefly described above are included in Fig. 6.1 which puts the manual, interactive, and automated capabilities into perspective. Note that the insertion of assertions (described briefly in Sec. 4 and in detail in the Functional Description) is shown as the first activity. The power of assertions lies in their ability to provide functional information about the program which both the test tool and user can analyze to determine correctness of program behavior and completeness in functional testing.

6.2 INSTRUCTION-LEVEL SIMULATION

With the advent of MIL-STD-1750, the military standard instruction set for airborne computers, it is not unreasonable to consider the incorporation of target machine requirements into a general-purpose, host-operational tool like J73AVS. Robert Glass at the Boeing Company has stressed the value of testing software on the host computer (see App. B). It is his contention that most errors in embedded systems can be traced to faulty code in the host computer. Further, it is only on the host system that computer and peripheral resources for extensive testing are available.

Simulation of the 1750 instruction set can be a functional component of J73AVS which contains default instruction size, precision, and cycle times for a typical target machine. The user can change the defaults through commands to represent actual processing requirements of his target. User-requested reports will provide simulated operational measurements for the target to determine if the software meets size, accuracy, and timing requirements.

6.3 CODE AUDITING

Code auditors for assessing the compliance of programs with certain standards are common software support tools. Although disciplined programming policies are encouraged, it is clear from high maintenance costs that such policies are not always followed. Computer

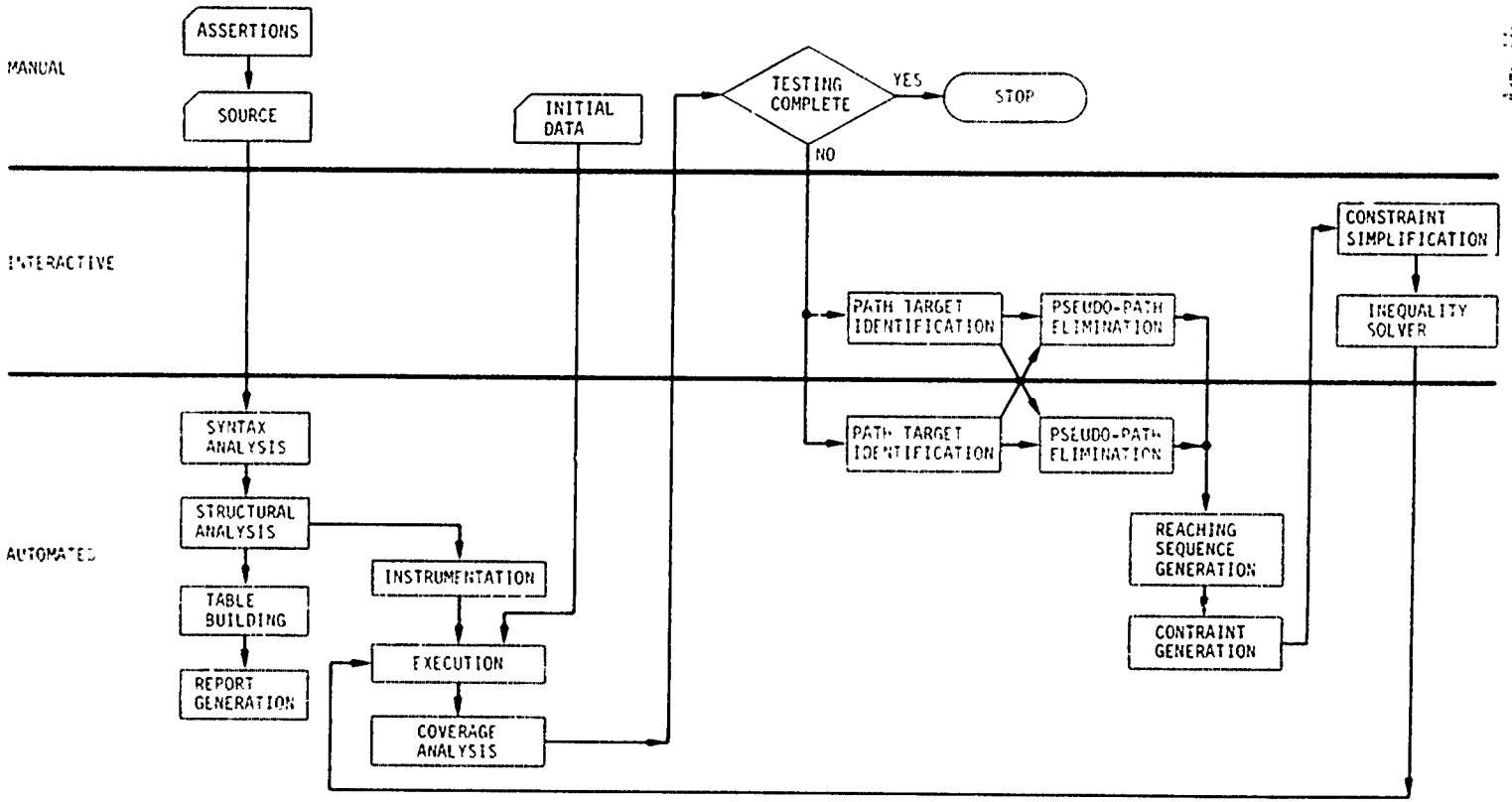


Figure 6.1. Operation of a J73AVS Test Data Generator

Sciences Corporation and TRW have used code auditors on both FORTRAN and assembly languages and have reported a formal cost reduction of $37,000 by using a FORTRAN code auditor on one project alone.[1] As soon as JOVIAL J73 has matured to a level where programmers can specify coding standards, they should be incorporated into the static analyzer component of J73AVS. The user would have the option to select the code auditing feature.

Typical, general coding standards include the following:

- Length of program units
- Nesting level of loops
- Calling arguments are not expressions
- In-line comments precede labeled statements, conditional statements, and invocations

6.4 UNITS CONSISTENCY

Requiring that each local variable and each global variable be specified in terms of the physical units it represents (if any) allows comprehensive checking of the consistency of units. This type of checking is particularly relevant to technical software where many physical properties are represented and there are many possibilities of confusion over units. Units can be checked on a multi-module basis if each module contains a description of the units for each physical variable it refers to. The form of the description for JOVIAL might be:

UNITS (<variable-list-1> = <units-expression-1>,
<variable-list-2> = <units-expression-2>,...)

[1] K. F. Fischer, "Software Quality Assurance Tools: Recent Experience and Future Requirements," Software Quality and Assurance Workshop, San Diego, November 1978.

An inconsistency in units is indicated if unlike units are added, subtracted, or compared. The physical-units analysis compares the right and left side of assignment statements, the right and left side of relational operations, and actual and formal parameters. For convenience in stating UNITS assertions, all constants are assumed to be unitless, except for zero, which will match any units expression. A variable is declared unitless by stating that its units expression is the constant 1, as in UNITS (PI = 1).

This capability is already available in GRC's SQLAB AVS for FORTRAN and Pascal. It is also recommended for inclusion into the MUST (Multipurpose User-Oriented Software Technology) program for HAL/S software.[1] This added static analysis could be incorporated economically by converting the existing method used in SQLAB. Violations of consistency would appear within the current J73AVS static analysis report (see the Functional Description).

6.5 EXECUTABLE ASSERTIONS PRECOMPILER

A minor effort to develop a JOVIAL J73 precompiler strictly for the purpose of translating logical assertions into executable JOVIAL J73 code would have major benefits in producing more reliable programs early in their development stage. The precompiler would exist as a JOVIAL J73 program that merely scans source code for ASSERT statements and translates them into several executable statements, including the TRACE directive, to report assertion violations.

An assertion precompiler would be more efficient than translating assertions to executable code by instrumentation, since the precompiler does not require the syntax and structural analysis and the database storage and manipulation needed by the multi-purpose J73AVS.

---

[1] R. N. Taylor, Integrated Testing and Verification System for Research Flight Software - Design Document, Boeing Computer Services Company, Feb. 1979.

APPENDIX A
LITERATURE SURVEYED FOR STUDY

Andrews,D. M., J. P. Benson, Advanced Software Quality Assurance, Softwrae Quality Laboratory User's Manual, General Research Corporation, CR-4-770, May 1978.

Barth, J. M., A Practical Interprocedural Data Flow Analysis Algorithm and Its Applications, University of California, Berkeley, May 1977.

Belford, P. C., Berg, R. A., Hannan, T. L., "Central Flow Control Software Develpoment: A Case Study of the Effectiveness of Software Engineering Techniques," 4th International Conference on Software Engineering.

Belford, P. C., Broglio, C., "A Quantitative Evaluation of the Effectiveness of Quality Assurance as Experienced on a Large-Scale Software Development Effort, " Software Quality and Assurance Workshop, San Diego, November 1978.

Benson, J. P., et. al., Software Verification : A State-of-the-Art Report, GRC, CR-1-638, Marcy 1978.

Boyer, R. S., Elspas, B., Levitt, K. N., Select--A System for Testing and Debugging Programs by Symbolic Execution," Submitted to the 1975 International Conference on Reliable Software, April 1975.

Brooks, N. B., Gannon, C., JAVS Jovial Automated Verification System, Vol. 3, General Research Corporation, CR-1-722, December 1976.

Brooks, N.B., Gannon, C., JAVS Jovial Automated Verification System, Vol. 2, General Research Corporation, CR-1-722/1, June 1978.

Clarke, L. A., "A System to Generate Test Data and Symbolically Execute Programs," IEEE Transactions on Software Engineering, Vol. SE-2, No. 3, September 1976.

Elspas, B., Green, M. W., Moriconi, M.S., Shostak, R. E., A JOVIAL Verifier, SRI International, January 1979.

Engels, G. J., Godoy, S. G., "Sneak Circuit and Software Sneak Analysis," Journal of Aircraft, Vol. 15, August 1978.

Fischer, K. F., "Software Quality Assurance Tools: Recent Experience and Future Requirements," Software Quality and Assurance Workshop, San Diego, November 1978.

Fosdick, L. D., Miesse, C., The Dave System User's Manual, Department of Computer Science, University of Colorado, March 1977.

Gannon, C., TAP Testing Coverage and Parameter Evaluation Program, General Research Corporation, November 1978.

Gill, C. F., Holden, M. T., "On the Evolution of an Adaptive Support Software System" AIAA Comptuers in Aerospace Conference 1977, Los Angeles.

Glass, R. L., Real Time Software Debugging and Testing: Introduction and Summary, The Boeing Company, September 1979.

Glass, R. L., Real Time Software Debugging and Testing: Definition of the Problem, The Boeing Company, September 1979.

Glass, R. L., Real Time Software Debugging and Testing: Proposed Solutions, The Boeing Company, September 1979.

Glass, R. L., JOVIAL J73 Software Quality Assurance Tools, Volume I - Introduction and User Manual, The Boeing Company, February 1979.

Glass, R. L., Software Reliability at Boeing Aerospace: Some New Findings, The Boeing Company, September 1979.

Goodenough, J.B., Gerhart, S. L., "Toward a Theory of Test Data Selection," IEEE Transactions on Software Engineering, Vol. SE-1, No. 2, June 1975.

Gutmann, M. J., Rang, E. R., Silverman, J. M., Verification Techniques for Flight Control Software, Honeywell Systems and Research Center, December 1978.

Holden, M. T. "The B-1 Support Software System for Development and Maintenance of Operational Flight Software," NAECON 1976 Record.

Holden, M. T., "Semi-Automatic Documentation of B-1 Avionics Flight Software Global Data," Naecon 1978 Record.

Howden, W. E., "Effectiveness of Software Validation Methods," Infotech: Software Testing, Vol. 2, 1979.

Howden, W. E., "An Evaluation of the Effectiveness of Symbolic Testing," Software - Practice and Experience, Vol. 8, 1978.

Hollowich, M. E., McClimens, M. G., The Software Design and Verification System (SDVS), TRW Defense and Space Systems Group, June 1976.

Howden, W. E., "Theoretical and Empirical Studies of Program Testing," IEEE Transactions on Software Engineering, Vol. SE-4, No. 4, July 1978.

Leach, D. M., Automated Test Case Generator - Final Technical Report, Logicon, September 1979.

Leach, D. M., Automated Test Case Generator - Functional Description, Vol. I - Self-Contained Version for General Algebraic Languages, Logicon, September 1978.

Leach, D. M., Automated Test Case Generator - System/Subsystem Specification, Vol. I - Self-Contained Version for General Algebraic Languages, Logicon, September 1979.

Maurer, W. D., "The Modification Index Method of Generating Verification Conditions," Proceedings of 15th Annual ACM S.E. Regional Conference, April 1977.

Miller, E. F., Jr., Methodology for Comprehensive Software Testing, General Research Corporation, CR-1-465, February 1975.

Moriconi, M. S., A System for Incrementally Designing and Verifying Programs, Vol. 1, USC/Information Sciences Institute, November 1977.

Ostrand, T. J., Weyuker, E. J., "Error-Based Program Testing," Presented at 1979 Converence on Information Sciences and Systems.

Ramamoorthy, C. V., et al. Fortran Automated Code Evaluation System (FACES) Part I, University of California, Berkeley, July '974.

Smith, K. A. "DAVE and PET at NASA Langley Research Center," Proceedings of VIM 31 Conference, Detroit, October 15-17, 1979.

Stickney, M. E., "An Application of Graph Theory to Software Test Data Selection," Software Quality and Assurance Workshop, San Diego, November 1978.

Stucki, L. G., et al, Software Automated Verification System Study, McDonnel Douglas Astronautics Company, January 1974.

Taylor, R. N., Integrated Testing and Verification System for Research Flight Software - Design Document, Boeing Computer Services Company, February 1979.

Preliminary Design Specifications SDVS Improved Conversational Language, TRW Defense and Space Systems Group, March 1978.

"NODAL, The Node Determination and Analysis Program," TRW Brochure, 1975.

"SURVAYOR, The Set-Use of Routine Variables Analysis Program," TRW Brochure, 1975.

APPENDIX B
REVIEW OF RELEVANT TECHNIQUES

JAVS Technical Report: Vol. 1 "User's Guide" 1975, 1976, 1978

JAVS Technical Report: Vol. 2 "Reference Manual" 1975, 1976, 1978

JAVS Technical Report: Vol. 3 "Methodology Report" 1976

Methodology for Comprehensive Software Testing 1975

JAVS Computer Program Documentation System Design and Implementation Manual 1975, 1976, 1978

JAVS Final Report 1975, 1976, 1978

General Research Corporation
Santa Barbara, California

The JAVS (JOVIAL Automated Verification System) and testing methodology were developed for the Air Force as a near-term solution to the problem of testing JOVIAL J3 software. The requirements for the tool were to provide an automated mechanism for measuring the thoroughness of testing and assisting with generating new test cases to increase the level of testedness. The resulting tool has the following functional capabilities:

1. Recognize JOVIAL J3 source text with very few language restrictions and build a database for up to 250 invokable modules with no limit on number of statements.

2. Using the database, identify potential structural infinite loops and unreachable code, insert software probes at each decision point, formulate software documentation reports showing symbol, statement, control path, module, and inter-module information.

3. When the instrumented modules are executed (with the remainder of the program, if the entire program is not instrumented), provide statement and branch coverage information and module execution timing data.

4. Provide lists of branches not executed by each test case and the sequences of branches required to be executed in order to reach the unexercised branches.

5. Provide an assertion language to assist code development and testing whereby user-supplied assertion statements can be converted to standard JOVIAL J3 by JAVS and supply execution time information.

JAVS does not provide data flow analysis capabilities for consistency checking, interface analysis, formal verification, or test data generation. The 1976 published methodology report provides guidelines for code development and testing which are keyed to the capabilities provided by the JAVS tool. The resources required by JAVS on the HIS 6180 are summarized below:

JAVS load size = 53K words

Data collection routines load size = 4K words

Random and sequential files

*Compile size of instrumented source = 15% larger than uninstrumented compile size

*Compilation time of instrumented source = 15% longer than for uninstrumented source

*Execution time of instrumented source = 50% longer than execution of uninstrumented source

*Coverage analysis time = 3-6 times execution time of instrumented source

---

* These resource requirements are rough estimates which vary according to the control structure of the program and coverage analysis options requested.

Integrated Testing and Verification System for Research Flight Software - Design Document

Richard N. Taylor
Boeing Computer Services Company
Seattle, WA
NASA Report 159008, February 1979

This design document describes a variety of software support tools to be included in the MUST (Multi-purpose User-Oriented Software Technology) system for HAL/S software. The tools included in this design operate from HALMAT, an intermediate representation of HAL/S. Thus, the tools do not have to perform any parsing. The types of tools are static analyzers, symbolic executors, and dynamic analyzers. There is heavy emphasis on static and dynamic assertion usage and statistics gathering.

The design recommendation is that small, modular facilities be combined in a variety of ways to accomplish program creation and maintenance. Such modular facilities are:

- local assertions
- regional assertions
- internal documentation
- answers about previously written code
- auditor
- units and scale checker
- cross-reference map generator
- data flow analysis
- execution-time monitoring
- instrumentor for run-time monitoring

Sample combinations of using these techniques are:

1. Isolating an error - dynamic analysis with extensive assertion usage on the suspect module.

2. Initial verification of new code - both data-flow and non-data-flow static analysis.

3. Broad-based verification with unlimited resources - static analysis, symbolic execution, test coverage.

4. Isolation of functional error - symbolic execution of appropriate paths, dynamic analysis.

5. Verification of previously verified modules - multi-purpose data-flow analysis and static checking of integration requirements, dynamic analysis of concurrent process characteristics.

Verification Techniques for Flight Control Software

E. R. Rang, J. M. Silverman, J. J. Gutmann
Systems and Research Center, Honeywell
December 1978

This report describes several manual and computer-assisted techniques for the verification of flight control software. "Verification" as used in this report means that the resulting system functions as intended. Therefore, the techniques described cover the description of requirements, specifications, design, testing, and assertion verification.

Flight control software has characteristics that distinguish it from other types of software. Among these characteristics are synchronization, distributed processing, assembly code, structurally simple functions, and simple data types. The verification techniques recommended in this report reflect these characteristics.

The techniques described and recommended are:

1. HIPO (Hierarchy plus input-process-output) charts
2. Formal specification using SRI's SPECIAL
3. Petri nets
4. Decision tables (as defined by Goodenough and Gerhart)
5. Symbolic execution

The HIPO charts provide a manual, disciplined method for stating software requirements, defining a system design, and, when used with decision tables, generating test data. HIPO charts allow for describing the system and its individual functions and can be used as a basis for design verification. Since the fabrication of HIPO charts is manual and there are no enforced standards for their thoroughness, their value is completely dependent upon the generator of the charts.

The use of HIPO charts facilitates drawing Petri nets, constructing decision tables, choosing test data, and performing manual symbolic evaluation of logical functions. Petri nets can be used to represent interacting concurrent processes, but they can become complicated very quickly. The primary asset of Petri nets is their usefulness in developing a preliminary design.

Decision tables consist of enumerating each decision (condition) in a program (C1, C2,...), followed by each action (A1, A2,...) to be undertaken, and then a set of test data (D1, D2,...) which will exercise each combination of conditions and alternatives (collectively called rules). Since HIPO charts include conditions and actions with the "process" section, decision tables can be generated easily. Theoretically, this technique of manual test data generation will exercise all sequences of conditions in a program. There are still two major problems: (1) if loops are involved, there may be an infinite number of condition sequences, and (2) if "moderate" data values are selected, errors can still exist which might otherwise be found by stress testing. As stated earlier, two of the characteristics of flight software are few loops and elementary data structures (frequently just boolean structures). For this software, then, a tool which automated the development of HIPO charts, translated them into decision tables, and generated the test data would be very beneficial.

SRI's approach to verification, as described in this report, is to formalize the software construction methodology, thus allowing machine-assisted verification. In their formal language, SPECIAL, a system is described before any considerations are included about implementation. Modules are formulated as finite-state automata: primitive data structures are the states, operations are the state transitions, and outputs are computed from the inputs and final states. The SPECIAL system is difficult to use, and the authors of this report were not convinced that the results were worth the trouble.

Symbolic execution is used along with user-supplied assertions to formally verify assembly code in the PLOVER-80 tool. The verification technique described in PLOVER-80 is similar to, but not quite as extensive as, that in SQLAB, a verification tool for FORTRAN and IFTRAN. PLOVER-80 accepts a set of assertion statements and the Intel 8080 assmbly code as input, internally generates inductive assertions with new variable names, and produces verification conditions using symbolic execution which must be manually proved to be correct.

The good feature of any assertion-based tester or verifier is that it offers an additional means of stating specifications in a module-readable form. We have found assertions to be extremely useful as execution-time checks during software testing. The bad feature of assertions is that they too can be erroneous, and if a proof of correctness relies solely on them, they had better be correct.

Boeing Support Software for Embedded Computer Systems - SCP

Purpose:

1. Generate loadable code
2. Support V & V
3. Support maintenance and configuration control

Capabilities:

1. Automated configuration management
2. JOVIAL/J3B compiler with multiple code generators
3. Generalized macro assembler with multiple targets
4. Generalized link editor supporting multiple targets
5. Specialized loaders supporting multiple target interfaces
6. Host computer statement level simulation
7. Multiple target computer instruction level simulations
8. Software version comparison at source, object, load levels
9. Automatic cross reference and flow chart generation

Design Concept:

1. Open-ended processor structure
   a. Table-driven common control program
   b. Single interface to host computer
2. Processors utilize common system routines
3. Processors interface through common database format
   a. Extensible data formats
   b. Database management utilities
4. Machine-independent processor design
   a. Preprocessors format machine-dependent tables
   b. Special processing routines may be added

5. Implementation in HOL
    a. AED used
    b. Machine dependencies isolated and parameterized

Documentation Processors:

1. Global cross reference
    a. Global data dictionary
    b. Storage allocation map
    c. Data block descriptions
    d. Procedure called-by/calls list

2. Flowchart
    a. Macro-level JOVIAL/J3B
    b. AP assembly language

Functional requirements for SCP are:

1. Modification of J3B cross-compiler to save descriptive and set/used information for data variables

2. Modification of the assembler to process operational software data and procedure coding conventions and to save descriptive and set/used information for data variables

3. Integrate the saved descriptive and set/used information with output of the linkage editor and source code comments to provide appropriate formatted listings

4. Allow text editing of resultant formatted listings

"Real Time Software Debugging and Testing: Proposed Solutions," Robert L. Glass, The Boeing Company, D180-25249-1, -2, -3, September 1979.

This report shows that most real-time software is tested in the target, not the host, computer environment even though there are no software checkout tools in the target environment. However, since more than half of the 20 projects surveyed in this report used HOL and since most errors are in the source code (not in generating the target's object code or in the target's environment), the emphasis of the proposed solutions is on the host computer environment. To check out the source code in the host environment, both the language debug facilities and a software environment simulator must be available on the host.

For the purpose of designing the J73AVS, only the debug and test proposed solutions (not those for an environment simulator) are critiqued. This set of recommendations can be summarized as follows:

1. Timing analysis can identify critical areas which should be recoded in assembly language.

2. Self-checking code, using conditional compilation, looks for input data acceptability, data storage overflow, assertions and range checking and provides for traces and dumps.

3. Data contention analysis can prevent timing errors due to parallel processing.

4. Audit trails of data and logic traces should be recorded.

5. Fault tolerance mechanisims provide for defensive programs.

6. A cross-reference listing should include structural relationships, data types, and set-used information. Both local and system-level cross-reference lists are needed.

7. Anomaly checking such as inaccessible code, undefined variables, type mismatching should be performed.

8. Structural testing should include logic branches, functions, and combinations of logic branches.

9. Data tracing, procedure tracing, and formatted snapshot dumping should be performed such that data is displayed by name, is properly formatted, and is tied to program structure.

10. Unsafe programming practices can be recognized, summarized, and reported.

Sneak Software Analysis
Boeing Aerospace Co.
Houston, Texas

Sneak analysis is a set of manual and computer-aided techniques for uncovering and predicting unplanned modes of operation. Given software code, reference manuals, requirements and specification, module descriptions, flow diagrams, data structure definitions, etc., as input, a manual encoding of the input is made. Outputs from the Sneak Software Analysis routines include: nodal set number report, variable name report, label name report, and mnemonic report. Certain questionable design practices are flagged such as unnecessary logic and unreferenced labels or variables. Then a manual verification process is undergone using the code, output reports, and specifications using a network tree representation of data and logic flow.

Of interest to AVS's are the set of clues accumulated through case histories:

		Implemented in J73AVS?
1.	Unused paths	Yes - dynamic analysis
2.	Inaccessible paths	Proposed for future effort (see Sec. 6.1)
3.	Improper initialization	Yes - static analysis
4.	Lack of data storage usage synchronization	Yes - performance analysis
5.	Bypass of desired paths	Yes - dynamic analysis
6.	Improper branch sequencing	Yes - dynamic analysis
7.	Potential undesirable loops	Yes - performance analysis
8.	Infinite looping	Yes - static analysis
9.	Unnecessary (redundant) instructions	Set-set-used detected
10.	Unreferenced labels	Yes - label report
11.	Bypassed variable initialization	Yes - static analysis

The Software Design and Verification System (SDVS)
TRW
Redondo Beach, California

SDVS is an integrated set of non-realtime software to aid in the development, coding, testing, and configuration management of avionics software (primarily DAIS, the AFAL Digital Avionics Information System). Its capabilities are: simulation of DAIS processors, automated configuration management of mission software, automatic control of simulation runs, editing and processing of data generated by the simulation, and a JOVIAL-like command language.

The command language provides statements for driving the simulation such as assigning values to variables, transferring control, collecting data, evaluating logical expressions, interpreting post-processing requests, formatting output, etc.

SDVS requires a J73/I compatible JOVIAL compiler and a database management system. It currently operates on a DEC-10.

The facilities for debugging and validating avionics software are:

1. Snapshot/rollback - during the course of a simulation, results are saved for a subsequent restart.

2. Data recording - statement, transfer, register, instruction traces; module execution clock times; values of selected variables traced; mod[illegible] data requested by user printed.

3. Post-simulation run processing - capabilities to sort, edit, analyze, and output simulation data.

Test Coverage Analyzer
Boeing Aerospace Corp.
Seattle, Washington

The JOVIAL J73/I Test Coverage Analyzer provides segment execution coverage analysis as an extension to the J73/I compiler. The extent of instrumentation: (a) all branch points, (b) all branch points and FOR loops, and (c) procedures only, is user-specified as a compiler control card option. Post-test analysis is performed by support and system routines, identified by the user at link time.

An example of the Test Coverage Analyzer's output is:

Procedure Name: APROC

Stmt. No.	Count	Stmt. No.	Count	Stmt. No.	Count
1	3	3	10	7	10
10	0	12	10	21	3

Procedure Name: DRIVER

Stmt. No.	Count	Stmt. No.	Count	Stmt. No.	Count
1	1	4	0	7	500
12	500	17	20	19	480
25	10	31	1		

The resource impact from using the Test Coverage Analyzer is:

1. Instrumented programs are 10-30% larger than uninstrumented programs. For procedures only, the overhead in size is 0-5%. The execute-time library is 1100 words.
2. Execution time is 40-60% longer for branch point analysis, 75-100% for branch point and FOR-loop analysis, and 10-30% for procedure analysis.
3. There is no significant size or time impact on the compiler.

The only limitation of the Test Coverage Analyzer is: no more than 1000 segments per compilation unit may be analyzed. This limitation may be easily increased.

A System for Incrementally Designing and Verifying Programs, Vol. 1
Mark S. Moriconi
University of Texas at Austin
and USC/Information Sciences Institute
NTIS Report ADA055501

This report, a doctoral thesis, presents a description and usage-by-example interactive dialogue of a verification system which differs from most other systems in two ways:

1. It supports software design and verification through incremental stages with minimal reprocessing of changed modules.
2. It provides a very friendly user interface with a responsive, hierarchical command language.

The system, called SID, is LISP-based and runs on a PDP-10 computer. Most of SID is written in Reduce; the rest is written in UCI-LISP.

The basic features of SID are to accept designs of modules in terms of assertions, determine what the unresolved external references are, and then automatically generate verification conditions (VC's). The system generates VC's for paths that are completely defined, ignoring those that are not. Thus, programs can be a mixture of specifications only, complete program text, or some in-between state of development. Verification is performed by an interactive theorem prover. Each VC is proved separately. When design changes are made, the system determines what new VC's need to be generated and proves only the new ones.

The aspects of SID that are interesting in the context of the J73AVS development are the system's determination of what has been changed in the software being analyzed and the conversational command

language. The SID commands are: Add, Delete, Edit, Explain, Help, Print, Prove, Restore, Save, Suggest, Translate, VCS, ?E,?,??. Most of the commands have subsequent levels of detail, prompting the user for more information as it is needed. As the Suggest and Explain commands imply, SID is capable of providing a certain amount of guidance for directing system activities and giving explanatory comments.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave pnysics and electronic reliability, maintainability and compatibility.